Resumen libro "Codigo limpio"

1. Introducción

Todos necesitamos una cosa cuando creamos codigo, la maestria, existen dos formas de conseguirla: conocimiento y trabajo, debemos adquirir los conocimiento de los principios, patrones, practicas propias de un maestro y aplicarlas a traves de la practica

aunque debemos tomar en cuenta que aunque sepamos todo para tener un Código limpio normalmente fallaremos, pero debemos seguir esforzandonos para poder llegar a hacer un buen codigo y llegar a la meta

---

Hágase el código

El código y los programadores no van a desaparecer ya que hay detalles de los requisitos que no se pueden omitir ni abstraer, se deben especificar para que un equipo pueda ejecutarlos

los que esperan que desaparezca es el quiere que una maquina sea creada para hacer lo que queremos en vez de hacer lo que les digamos, y esto necesitaria que las maquinas lograran traducir necesidades ambiguas en programas perfectamente ejecutados que logren satisfacer nuestras necesidades a la perfeccion

eso es imposible que suceda ya que ni siquiera un programa ha podido satisfacer las necesidades de un cliente

Necesitamos el código porque este nos ayuda a expresar los requisitos en ultima instancia, logramos crear lenguajes que logren asemejar los requisitos, podernos crear herramientas para analizar y combinar los requisitos en estructuras formales, pero no pueden hacer una precision necesaria

Codigo Incorrecto

Es necesario tener un buenc codigo porque a la hora de tener que modificarlo puede llegar a tener fallos si se quiere hacer algo con el, hace un tiempo una empresa famosa creo y distribuyo un programa con un codigo erroneo y al intentar agregar funciones el codigo fallaba y esto llevo a la empresa a desaparecer

si eres programador normalmente te has encontrado con un codigo dificil de entender o mal organizado y esto ha hecho que sea complejo trabajar con el, pueden haber diversas causas de esto, falta de tiempo, querer acabar con el programa rapidamente

en el momento que vemos lo dificil que puede ser lo posponemos y nunca lo arreglamos, si el programa a pesar de mal hecho que este construido funcione, lo dejamos asi y no nos preocupamos por lo que pueda pasar en el futuro, segun la ley de LeBlac: "Despues de igual a nunca".

El coste total de un desastre

Entre mas experiencia tengas programando mayores seran los problemas que afronte cuando trabaje en equipo al realizar el codigo, cualquier cambio hecho en el mismo puede afectar a todo el programa, se debe tener en cuenta los detalles, efectos y consecuencias

Mientras los problemas aumenten paralelamente la productividad disminuye, al intentar solucionar esto puede traer mayores problemas a los que habia anteriormente

---

El gran cambio de diseño

En el momento que se dan cuenta que esto cada vez cae mas el grupo no quiere trabajar mas en el codigo y quieren que se cambie de direccion, evaluan que un cambio de diseño en el proyecto seria contraproducente pero no pueden ignorar la constante caida de la productividad

En ese momento se debe aceptar el cambio pero todos quieren formar parte de ello porque es un lienzo en blanco y dicen que podrian plasmar todas sus ideas en el, pero en ese momento solo se deben elejir a los mejores para el correcto funcionamiento

para esto se deben tener en cuenta los dos sistemas, se debe tomar en cuenta el antiguo para crear el nuevo, esto no cambiara hasta que el nuevo sistema haga lo que el antiguo hacia

Esta competicion puede durar mucho tiempo pero en el momento que se decide no hay ningun antiguo programador y los nuevos estan exijiendo lo mismo que los antiguos por el mismo problema, y asi el ciclo se repite

por esto desde el inicio debemos hacer un buen codigo para no caer en este bucle

---

Actitud

En la vida como programadores podemos encontrarnos con problemas que parecen ser faciles pero pueden consumir mucho mas tiempo del que esperabamos, intentamos echarle la culpa a cosas externas cuando el problema podemos ser nosotros

En el proceso de creacion de un proyecto normalmete nos vamos a encontrar con mas personas que nos exijen cosas especificas del proyecto pero en eso no damos nuestra opinion y eso puede ser perjudicial a futuro, cargamos con la responsabilidad del proyecto, mas cuando sale algo mal por el codigo

tenemos miedo que si damos una opinion contraria seamos despedidos, pero el trabajo del director es defender los objetivos y nosotros debemos defender nuestro codigo como si fuera lo unico que nos importa

no debemos ceder a las exijencias de un jefe a pesar que eso no sea profesional, los programadores son los unicos que saben todos los aspectos de la implementacion de alguna cosa para evitar un posible desastre

---

El enigma

Los programadores deben enfrentar el enigma de hacer su Código lo mejor posible porque saben que cualquier error los puede llevar a atrasar el trabajo de una forma inmediata y hace que no puedan llegar a los plazos de entrega

¿El arte del código limpio?

La unica forma de evitar desastres significativos es tener un codigo limpio pero es complejo llegar a tener un buen codigo limpio, debemos saber como crearlo, no simplemente saber evaluar si un codigo es limpio o no

para lograr hacerlo debemos dominar muchas tecnicas y tener un amplio sentido de correccion en nuestro codigo, si logramos obtener esto podemos llegar a ser como un artista que puede formar un lienzo en blanco en un sistema de codigo correcto

Los programadores que no logran adquirir este sentido pueden detectar problemas pero no dar una solución al mismo

Concepto de Código limpio

Existen tantas definiciones como programadores

Bjarne Stroustrup, inventor de C++ y escritor de "The C++ Programming Language"

El decia que para que su codigo fuera elegante y eficaz debia tener:

- Logica directa: esto ayuda a evitar errores ocultos
- Las dependencias deben ser minimas: esto ayuda al mantenimiento
- rendimiento optimo: para que los usuarios no tiendan a estropear el codigo

El dice que el codigo debe ser agradable a la vista a la hora de leerlo

Bjarne menciona que todos los detalles son importantes para que un codigo sea codigo limpio

Por ultimo menciona que para considerar un código "limpio" debe ser concreto, todas las funciones, clases, módulos deben tener una actitud invariable y que no se contamina por detalles circunstanciales

Grady Booch, autor de Object Oriented Analysis and design with applications

El tiene la misma perspectiva de Bjarne ,el dice que el código limpio debe poderse leer bien, como un libro, debe poderse ver claramente con ese programa soluciona perfectamente el problema, debe ser especifico y no especulativo, solo debe incluir lo necesario

"Big" Dave Thomas, fundador de OTI, el padrino de la estrategia Eclipse

Thomas dice que el código debe ser lo mas claro posible, solo puede tener una forma de conseguir su objetivo y solo debe tener lo estricto y necesario y nada más para que el mismo o otros programadores pueda mejorarlo

Thomas dice que el código no puede ser "limpio" sin pasar por una fase de limpieza de pruebas, "no importa si el código es lo mas elegante, legible y accesible, si no es probado, no es limpio"

Michael Feathers, autor de Working Effectively with Legacy Code

Michael dice que existe una cualidad principal sobre las demás, el código limpio siempre parece ser escrito por alguien a quien le importa que sea limpio, el habrá mejorado su código lo máximo posible

Ron Jeffries, autor de Extreme Programming Installed y Extreme Programming Adventures in C#

El dice que el código limpio debe tener los mínimos duplicados, es decir, que en el momento que encontremos que algo se repite muchas veces debemos ver cual es y expresarla con una mayor claridad, debe maximizar la expresividad y diseñar sencillas abstracciones en las fases iniciales

Ward Cunningham, inventor de Wiki, Fit y uno de los creadores de la programación eXtreme. Uno de los impulsores de los patrones de diseño. Una de las mentes tras Smalltalk y la programación orientada a objetos. El padrino de todos los que les importa el código

El dice que el código limpio debe ser legible y lo mas sencillo posible, que parezca que el lenguaje hubiera sido diseñado solo para la solución del problema

dice que la practica del código limpio debería ser lo normal, no esperar un código complicado y desordenado el cual haga complejo extraer el razonamiento del sistema.

Escuelas de pensamiento

En este libro nos van a presentar lo que para los autores es código limpio, variables, funciones, clases de una forma absoluta, desde su experiencia

nos explican que depende del educador podemos creer que es la correcta y solo nos concentraremos en ella hasta lograr aprenderlo todo

Somos autores

Debemos recordar que el código que nosotros escribamos también va a leerlo otras personas o nosotros en un futuro, por eso debemos tener nuestro código lo mejor posible, esto también ayuda que sea mas rápido la creación del código que estemos haciendo

La regla del Boy Scout

Debemos aplicar la regla de los Boy Scout en nuestro día a día programando

> Dejar el campamento mas limpio de lo que se ha encontrado.

Nuestro labor sera mejor si aplicamos esta regla, no dejamos que el código se corrompa y sera más fácil si se llegara a reutilizar

Nombres con sentido

Usar nombres que revelen las intenciones

En el momento en el que queremos nombrar una variable, clase o funcion debe tener los siguientes tres aspectos

- Indicar porque existe
- que hace
- como se usa
- si requiere un comentario significa que no revela su contenido

Un ejemplo de esto es la diferencia entre declarar

```
int d; //tiempo transcurrido en dias
```

y

```
int elapsedTimeInDays;
```

el segundo es mas descriptivo con su función y hace que sea mas fácil su lectura

ahora en una funcion

```
public list<int[]> getThem() {
  list<int[]> list1 = new ArrayList<int[]>();
  for(int[] x : theList)
    if (x|0| == 4)
      list1.add(x);
```

```
return list1
}
```

y este codigo

```
public list<cell> getFlaggedCells() {
  list<cell> flaggedCells = new ArrayList<cell>();
  for(cell cell : gameBoard)
    if (cell.isFlagged())
      flaggedCells.add(cell);
return flaggedCells
}
```

Como se puede ver en los dos codigos solo con cambiar el nombe podemos hacer mas descriptivo y facil de entender nuestro codigo

Evitar la desinformación

Cuando estemos definiendo los nombres debemos evitar siempre dejar pistas falsas que dificulten el significado del mismo, por ejemplo no debemos poner variables como hp, hp puede parecer ser la abreviatura de hipotenusa pero puede no serlo

No debemos hacer módulos que se parezcan a otros, porque puede confundir al que lo esta leyendo, por ejemplo XYZControllerForEfficientHandlingOfString y XYZControllerForEfficientStorageOfStrings

en estos dos nombres si se lee por encima parecen las mismas variables y esto es lo que debemos evitar

Realizar distinciones con sentido

En el momento que se necesite usar elementos distintos pero que están relacionados en el mismo ámbito

si es numérico lo mejor es hacer una variable tipo array con un for para adicionar elementos a la variable y se logre entender mas fácil

```
public static void copyChars(char a1[], char a2[]){
  for(int i = 0; i < al.length; i++){
    a2[i] = a1[i];
  }
}
```

se deben diferenciar los nombres de forma que el lector aprecie las diferencias

Usar nombres que se puedan pronunciar

En el momento de crear los nombres debemos tener en cuenta crear nombres pronunciables para que sea mas facil explicarlo

es diferente tener una clase

```
class DtaRcrd102 {
  private Date genymdhms;
  private Date modymdhs;
  private final String pszqint = "102";
  /* ... */
};
```

a esta clase

```
class Customer {
  private Date generationTimestamp;
  private Date modificationTimestamp;
  private final String recordId = "102";
  /* ... */
};
```

el segundo directo es mas descriptivo y es mas fácil explicarlo a personas nuevas

Usar nombres que se puedan buscar

En el momento que debamos escoger los nombres de las variables debemos tener en cuenta que en el momento que necesitemos buscar una variable especifica debemos ser claros, es decir, es diferente tener que buscar una variable E en un código amplio a buscar la variable MAX_CLASSES_PER-STUDENT

Si es una variable que se usa en varios puntos del código lo mejor es asignarle un nombre que se pueda usar y si es una variable local lo mejor es usar nombres de una letra

Evitar codificaciones

debemos evitar lo mayor posible codificar la informacion, esto hace que sea mas tardato y añade dificultad al momento de invocarlas, esto puede afectar a los nuevos programadores por la mayor e innesesaria carga mental al trabajar con el codigo

Notación Húngara

En el pasado cuando se usaba fortran y Basic era mas complejo hacer código, antes se debia tener mucho cuidado al nombrar funciones porque en fortran se convertia la primera letra de un tipo en codigo y en Basic solo se permitia una letra y u numero, por lo que era mas complejo programar

A dia de hoy los compiladores recuerdan los tipos y los aplican, los lenguajes disponen de sistemas de tipos mas complejos

Prefijos de miembros

Lo prefijos ya no son necesarios a día de hoy ya que conforme se va aprendiendo los programadores los omiten cada vez mas y solo se fijan en la parte con sentido del nombre

Interfaces e implementaciones

hay un caso especial para usar codificaciones, en el momento que se necesite crear una interfaz, es mejor usar ShapeFactoryImp que IShapeFactory para evitar que los usuarios sepan que se trata de una interfaz

Evitar asignaciones mentales

Debemos evitar que las personas que lean nuestro codigo deban traducir los nombres de variables por otro que conozcan

debemos evitar poner nombres como a, b, c, entre otros porque eso da la sensación de que nuestro código es pobre

Nombres de clases

los nombres nunca pueden ser un verbo, en vez de llamar a una variable Manager, Processor, Data y Info, es mejor llamarla Customer, WikiPage, Account y AddressParser

Nombres de metodos

Los nombres de los metodos deben tener nombres de verbo como deletePage o save. Si son metodos de acceso, modificacion y predicados debe tener como nombre su valor y un prefijo como get, set e is, como por ejemplo

```
string name = employee.getName();
customer.setname("mike");
```

en el momento de usar constructores es mejor usar metodos de factoria estaticos con nombres que describan los argumentos. Por ejemplo:

```
Complex fulcrumPoint = Complex.FromRealNumber(23.0);
```

es mejor que

```
Complex fulcrumPoint = new Complex(23.0);
```

No se exceda con el atractivo

Debemos ser claros al poner nombres, no debemos intentar ser graciosos al nombrarlas, por ejemplo es mejor para el lector que una función se llame abort() que eatmyShorts()

Una palabra por concepto

Al momento de elegir los nombres debemos elegir una palabra por cada concepto abstracto y mantenerla, si esto no se hace hara mas complejo y tardado al buscar a que se refiere esa función

Un lexico coherente ayuda a los programadores que tengan que usar su codigo

No haga juegos de palabras

Nosotros como autores debemos facilitar la lectura del código, debemos evitar la misma palabra con dos fines distintos, si tenemos una funcion que sume dos valores existentes y otra que añada su parametro a una coleccion lo mejor es llamar a una add y a la otra insert o append para que se puedan diferenciar y hacer mas efectiva la lectura

Usar nombres de dominios de soluciones

lo mejor es usar términos informáticos, algorítmicos, nombres de patrones, términos matemáticos, entre otros, ya que normalmente los que van a leer nuestro codigo van a ser programadores y deben entender este tipo de terminos, por ejemplo, para un programador le es mas facil entender el nombre "AccountVisitor" que "VISITOR"

Usar nombres de dominio de problemas

Si no existe un termino de programación para lo que esta haciendo es mejor usar el nombre de dominio de problemas para que el programador pueda preguntar el significado a un experto en dominios

Añadir contexto con sentido

En la mayoria de nombres debemos poner un contexto para que se logre entender mejor, en clases, funciones y espacios de nombres con nombres adecuados, si esto falla podemos recurrir a prefijos como ultimo recurso

si exitste variables como firstname, lastname,street, houswnumber, city, state, y zipcode ,se puede intuir que forman una direccion, lo mejor es crear funciones por separado ý no dejarlas sueltas para que se entienda que estan en un contexto

No añadir contextos innecesarios

Lo mejor es saber donde debemos poner los prefijos y saber si son demasiado largos para poder hacer que el codigo sea mas facil de leer, por ejemplo, Si queremos poner los nombres accountAddress y customerAddress es mejor para una instancia de clase Address que para un nombre de clase

Capitulo 3: Funciones

Tamaño reducido

debemos tener funciones lo mas cortas posible, esto ayuda a facilitar la lectura del código, recomienda que el código tenga una longitud aproximada de 20 líneas

Bloques y sangrado

Las funciones no deben pasar de una linea de codigo y debe ser solo para invocar una funcion, esto ayuda a que sea mas descriptivo y facil de leer}

Hacer una cosa

debemos ver que las funciones que creamos solo debe hacer lo que dice su nombre y debe tener el menor nivel de abstracción posible

Secciones en funciones

Las funciones que se dividen en secciones como declaraciones, inicializaciones y filtros se pueden considerar como funciones que hacen mas de una cosa, debemos tener funciones que no se puedan dividir en secciones

Un nivel de abstraccion por funcion

el nivel de abstraccion de una funcion debe ser el mismo dentro de la misma, no se puede tener un getHTML(); y un .append(); dentro de la misma funcion, esto hace que la funcion quede confusa y dificil de entender

Leer codigo de arriba a abajo: la regla descendente

debemos escribir codigo ordenado descendentemente por niveles de abstraccion, es decir, que a medida que vamos leyendo el codigo vaya bajando el nivel de abstraccion de las funciones

instrucciones Switch

debemos evaluar si es mejor usar una instruccion switch o mejor usar una funcion, ya que tener un switch hace que el codigo sea mas largo y ya que un switch la mayoria de veces hace mas de una cosa sera mas dificil leer el codigo

lo que debemos hacer es ocultar el switch con metodos abstractos para que nadie la vea

Usar nombres descriptivos

Cuando vamos a nombrar una funcion lo mejor es tenet el nombre lo mas descriptivo posible sin importar su longitud, es decir, que entre mas concreto sea el nombre de la funcion mejor

Argumentos de funciones

debemos usar la menor cantidad de argumentos en nuestro codigo, esto afecta a los lectores a la hora de entender el codigo, entre mayor sean los argumentos de una funcion peor sera el codigo

Formas monódicas habituales

cuando vamos a pasar parametros en una funcion debemos ser coherentes con la funcion, es decir que el parametro y el nombre no tengan nombres que no esten relacionados

Argumentos de indicador

Nunca debemos usar variables booleanas en una funcion, hace que codigo sea mas dificil de entender y indica que la funcion hace mas de una cosa, como se menciono anteriomente lo mejor es que una funcion solo debe hacer una cosa

Funciones diadicas

debemos eitar lo maximo posible usar las funciones con dos parametros, es cireto que hay funciones que obligatoraimente usan dos parametros como al usar el plano cartesiano pero si encontramos una funcion asi debemos ver si se puede simplificar mas por ejemplo convertirlas en metodos o crear clases que usen el otro parametro en su constructor

Triadas

Si las funciones con doble parametro son dificiles de entender y debemos evitarlas estas no se deben usar casi nunca, solo en momentos muy concretos

Objeto de argumento

Si una funcion parece tener dos o mas variables podemos evaluar si deberiamos pasarlo por una sola creando objetos cuando las variables esten en un mismo contexto

Listas de argumentos

Cuando se usan argumentos tipo string y se procesan de la misma forma los mejor es usar un solo argumento tipo List, debemos ver que si usamos esto no debe superar los 3 argumentos porque seria un error

Verbos y palabras clave

Cuando decidimos el nombre de los parametros lo mejor es usar nombres que tengan que ver con el nombre de la funcion, es decir, si nombramos a una funcion assertExpectedEquals lo mejor es usar parametros como Expected y actual para que se entienda mejor la funcion

Sin efectos secundarios

Debemos ver que nuestra funcion no tenga efectos secundarios, es decir, que nuestro codigo no hagan cosas ocultas ya que una función solo debe hacer una cosa

Argumentos de salida

Debemos evitar los argumentos de salida, antes de la programacion orientada a objetos era necesario escribir report.appendFooter(); y esto hacia que leer la funcion requiriera un esfuezo doble para entenderla pero ahora solo debemos invocar appendFooter();que hace ,mas facil entender el objetivo de la función

Separacion de consultas de comando

Las funciones pueden tener dos caminos, debe hacer algo o responder a algo pero nunca hara las dos, hacer esto puede hacer que sea mas dificil entender lo que hace la funcion, no sabemos que queria decir el autor con esa funcion, si nos encontramos con este tipo de funciones lo mejor es separar el comando de la consulta para evitar la ambiguedad

Mejor excepciones que devolver códigos de error

Si necesitamos devolver codigos de error no debemos hacerlo en funciones porque genera estructuras anidadas y al momento de generar el error el invocador lo genera de forma inmediata, lo mejor es usar el bloque try \ catch para separar el codigo de ruta y se simplifica mas

Extraer bloques Try\Catch

Cuando debemos usar una funcion try\catch lo mejor es separar las dos funciones y convertirlas en individuales

El procesamiento de errores es una cosa

Cuando separamos la función try\catch no deben hacer mas de una cosa, es decir, que si usamos la palabra clave try debe ser la primera de la función y no debe hacer nada mas después del bloque catch

El imán de dependencias Error.java

Cuando necesitamos hacer devolucion de codigos lo normal es crear una clase con todos los errores pero los programadores no la cambian porque implicaria volver a compilar todo, entonces lo que hacen es reusar codigo viejo

Lo mejor en esos casos es usar excepciones así se puede añadir nuevos errores sin la necesidad de volver a compilar

No repetirse

Si vemos que usamos mas de una vez una funcion dentro de una clase lo mejor es usar el metodo include para evitar usar lineas de codigo de mas

Programacion estructurada

Muchos programadores usan la regla de la programacion orientada a objetos

> todas las funciones y bloques deben tener una entrada y salida

Aunque para que la función sea mas descriptiva lo mejor es usar instrucciones como return, break o continue

Como crear este tipo de funciones

Cuando vayamos a crear una función lo primero que debemos hacer es tener un borrador el cual normalmente sera extensa y compleja, con extensas listas de argumentos, código duplicado entre muchas otras cosas que hace muy complejo entenderlo

pero despues debemos aplicar lo que vimos en lo anterior, retocar el codigo, dividir funciones, cambiar nombres, ente otras cosas y quedara el mejor codigo legible

Capitulo 4: Comentarios

debemos saber en que momento usar un comentario para tu codigo porque normalmente el codigo se vuleve obsoleto y desactualizado, lo mejor es que podamos contar perefectamente la funcion de nuestro codigo sin la necesidad de comentarios

Los comentarios no compensan el codigo incorrecto

Si creemos que el codigo necesita comentarios lo mejor es que se dedique a limpiar el codigo para que sea mas expresivo y facil de entender

Explicarse en el codigo

Debemos ser capaces de expresar las intenciones de nuestro codigo sin recurrir a comentarios, por ejemplo es mejor tener una funcion que explique lo que hace el codigo

```
if(employee.isEligibleForFullBenefits())
```

que tener un comentario con su funcion

```
// Comprobar si el empleadotiene derecho a los beneficios
if((employee.flags & HOURLY_FLAG) && (employee.age > 65))
```

Comentarios de calidad

Depende del caso pueden existir buenos comentarios en el código, pero lo mejor es no hacerlos

Comentarios informativos

En ocasiones hacer comentarios con el resultado de un método es útil pero lo mejor sera hacer funciones y que el nombre explique el resultado del mismo

Explicar la intencion

Podemos usar comentarios para detallar nuestra decision de porque lo escribimos así

Clarificacion

Podemos usar los comentarios para traducir el significado de un argumento o valor para que sea mas facil entenderlo, esto es conveniente usarlo cuando no se pueda alterar el codigo o cuando forma parte de una biblioteca, pero si usamos incorrectamente el comentario puede hacer mas dificil el entender el codigo, asi que debemos evaluar si es correcto el comentario o existe una mejor forma de solucion

Advertir de las consecuencias

Podemos usar comentarios cuando damos explicaciones a otras personas que lean nuestro codigo de porque algo no esta funcionando, esto sirve para advertir de no usarlo y su posible solucion por si quiere mejorarlo

Comentarios TODO

En ocasiones los programadores usan los comentarios TODO para tener un recordatorio de lo que deben modificar, eliminar, entre otras cosas

Aunque es muy util debemos recordar que no podemos sobrecargar el codigo con este tipo de comentarios, lo mejor es resolver la mayor cantidad posible

Amplificación

Podemos usar comentarios para darle importancia a algo que parecia no tenerlo

javadoc en API publicas

Cuando se usa una API publica lo mejor es que este bien comentada para entender las funciones rápidamente

Comentarios incorrectos

La mayoria de comentarios son de codigo mal hecho, como si el programador se hablara asi mismo

Balbucear

Cuando escribimos un comentario debemos estar seguros que sea lo mas explicativo posible, no podemos dejar que el comentario "quede de adorno" o que no se entienda lo que quiere dar a explicar

Comentarios redundantes

Debemos poder ver si un comentario en verdad ayuda a entender el codigo, porque puede existir comentarios que no ayuden al codigo, sino que lo perjudican haciendo mas dificil leerlo, lo mejor es si el codigo tiene lo que se ha mencionado anteriormente no usar comentarios

Comentarios obligatorios

No existen los comentrios obligatorios, nunca debemos colocar un comentario a no ser que sea estrictamente necesario

Comentarios periodicos

En la actualidad los comentarios periodicos son innecesarios y estorban en el codigo ya que hoy en dia existen buenos sistemas de control de codigo fuente, lo mejor es eliminarlos

Comentarios sobrantes

En el codigo existen comentarios que no aportan nadad al codigo, lo mejor es pulir el codigo lo mejor posible antes que esforzarse a poner un comentario

Comentarios sobrantes espeluznantes

En el momento de hacer comentarios no pueden usar comentarios para describirlo too y que al final sean comentarios sin sentido

No usar comentarios si se puede usar una funcion o una variable

Si hay comentarios que puedan pasarse a codigo lo mejor es hacerlo, elimina la necesidad de comentarios y hace mas legible el codigo

Marcadores de posicion

debe saber en que momento usar un marcador de posicion, usarlos en exceso resulta dser ignorado en el y lo mejor es ver si es prioritario y necesario el uso de los mismos

Comentarios de llave de cierre

Solo debemos usar comentarios en la llave de cierre cuando sea una funcion demasiado extensa con estructuras anidadas, de lo contrario, no es necesario y estorbaria

Asignaciones y menciones

Nunca uses asignaciones y meciones en tu codigo, a dia de hoy eso ya se encarga el sistema de control de flujo

Codigo comentado

¡JAMAS! debemos usar código comentado, si necesitamos cambiar algo, lo mejor es usar el sistema de control de flujo

Comentarios HTML

Los comentarios en codigo HTML si se usa una herramienta como Javadoc los programadores no deben entrar en ellos, solo arruinan la lectura

Informacion no local

Cuando usemos comentarios para describir la salida de un metodo debemos hacerlos con codigo que le rodee, no podemos hacer comentarios en el principio del archivo que describa el ultimo metodo de todo

Demasiada informacion

Debemos ser concretos cuando hacemos comentarios, no podemos dar informacion irrelevante al lector porque no es necesario

Conexiones no evidentes

Al usar comentarios deben tener una conexion evidente, no podemos dar una explicacion en el comentario y que en el codigo se entienda una cosa distinta

Encabezados de funcion

Lo mejor es no usar comentarios antes de una funcion, si esta con un nombre explicativo y solo hace una cosa no es necesario usarlos

Javadocs en codigo no publico

Cuando hacemos Javadocs que nos on publicos no podemos saturar el codigo de comentarios, a parte de ser innecesario, es molesto para el lector, el mejor comentario es el que se puede hacer un metodo

Formato

La función del formato

Lo mas importante de un codigo, fuera del funcionamiento, es la legibilidad, nuestro codigo debe ser los mas facil de leer posible, esto nos ayudara en el futuro si queremos cambiar algo lo hara mas rápido

Formato vertical

Cuando escribimos codigo debemos fijarnos en la longitud del mismo, lo mejor es escribir codigo en archivos que no superen las 500 lineas de codigo y no sea menor a 200

La metáfora del periódico

Nuestro codigo debe tener una estructura ordenada, como si fuera un periodico

Primero debemos tener un nombre que describa lo que nos vamos a encontrar, despues debemos encontrar los elementos ordenados descendientemente de mayor a menor complejidad y deben aumentar los detalles al seguir leyendo

Apertura vertical entre conceptos

Debemos separar por una linea en blanco cada concepto del código para mejorar la legibilidad

Densidad vertical

Si hay funciones o conceptos relacionados lo mejor es dejarlas estrechamente unidas, no podemos poner entre ellas cosas como comentarios innecesarios

Distancia vertical

Las clases y variables relacionadas deben estar lo mas juntas posible en un mismo lugar para su mejor comprensión

Declaraciones de variables

Cuando declaramos una variable siempre debe ser en la parte superior de los métodos como en todo el archivo

Variables de instancia

Las variables de instancia, dependiendo del lenguaje se suelen colocar en el inicio o fin de una función, no importa donde se escriban, solo deben estar en un lugar conocido para que sea mas fácil encontrarlas

Funciones dependientes

Debemos tener un orden al momento de invocar entre funciones, cada función debe estar lo mas cerca posible de la otra, esto facilita la detección de las funciones usadas y mejora la lectura del mismo

Afinidad Conceptual

Si vamos a usar un mismo tipo de variables en diferentes métodos lo mejor es agruparlas lo mayor posible

Orden vertical

Si vamos a hacer una invocación lo mejor es que primero este la función a invocar y después donde se invoca

Formato horizontal

Lo mejor es que una linea en un formato normal no sobrepase la pantalla, pero el limite se puede decir que esta entre 80 y 120 caracteres

Apertura y densidad horizontal

debemos saber si un elemento esta directamente relacionado o tiene una relacion menos estrecha, si lo tiene los teminos deben estar juntos, si no se le debe dar un espacio de distancia para mejorar la comprension

Alineacion horizontal

Nuestro codigo no debe tener alineacion horizontal, a parte de hacer mas dificil de leer el codigo al final las herramientas automaticas los eliminan

Sangrado

El sangrado nos ayuda a darle una jerarquia a nuestro codigo y llevan el siguiente orden

Declaraciones de clase

    metodos

        Implementaciones de bloques

Esto ayuda a poder leer mejor el codigo, sin el sangrado, los programas serian ilegibles

Romper el sangrado

Nunca debemos romper el sangrado, si vemos ámbitos replegados a una linea lo mejor es aplicar el sangrado

Ambitos ficticios

A veces al usar un for o while su cuerpo es ficticio, si se llega a usar este metodo lo mejor es poner el punto y coma en una linea distinta

Reglas del equipo

Si estamos en un equipo debemos definir una única forma de programar, esto le da un sentido y coherencia al programa

Objetos y estructuras de datos

Abstraccion de datos

Una interfaz debe ser abstracta, es decir que no se debe saber para que son, para hacer esto se necesita que las operaciones sean atómicas(Al momento de leerse la variable se ejecuta la operación paralelamente)

Antisimetria de datos y objetos

Debemos saber si las clases deben esconder su implementación interna o solo mostrarlos sin importar nada

La ley de Demeter

La ley de Demeter afirma que un método solo puede ser invocado por:

- El mismo
- Una variable dentro del metodo
- Un argumento
- En una variable de instancia del método

Y estos no deben ser usados en otros métodos

Choque de trenes

La ley de Demeter solo aplica a objetos, no a estructuras de datos, lo mejor es dividir el código cuando se necesite

Híbridos

Cuando hay un choque de trenes se forma un hibrido entre estructura de datos y objeto, estos deben ser evitados ya que dificultan la inclusion de nuevas funciones

Ocultar la estructura

Debemos tener claro si el método es estructura de datos o objeto , si es un objeto debemos indicar que haga algo, si es estructura de datos preguntar por sus detalles internos, si se mezcla distintos niveles de detalle no cumpliría la ley de Demeter y la base de datos lea objetos que no debería conocer

Objetos de transferencia de datos

El Data Transfer Object o OTD es un objeto que transporta datos entre procesos, estos son utiles para estructuras para comunicarse con bases de datos o analizar mensajes de conexiones

Registro activo

El registro activo es una forma de OTD que usa variables publicas pero tienen métodos de navegación como save y find , este es útil pero hay programadores que al

usarlo crean híbridos y esto es malo para el código, lo mejor es separar el registro activo a una estructura de datos y hacer objetos independientes que oculten sus datos internos

Procesar errores

Usar excepciones en lugar de códigos devueltos

Antes para capturar errores era limitado, se debia definir un indicardor de error o se devolvia un codigo de error que el invocador pudiera comprobar

esto trae un problema y hace que el invocador se confunda, es mejor generar una excepción al detectar un error

Crea primero la instrucción try-catch-finally

Al momento de hacer código lo mejor es crear los bloques try-catch lo mejor es primero intentar generar el error para capturarlo y lograr crear correctamente la excepción

Usar excepciones sin comprobar

Debemos evaluar si es necesario usar excepciones try/catch, esto puede ser benefioso pero si se usa debemos tener en cuenta que también debemos hacer cambios en los niveles inferiores del código, esto puede ser útil si se necesita crear una biblioteca critica, pero si es en desarrollo de aplicaciones generales los costes de dependencia superan las ventajas

Ofrecer contexto junto a las excepciones

Si vamos a usar excepciones lo mejor es redactar cual es el error y pasarlos junto a las excepciones para saber su origen y ubicación y registrarlo en el catch

Definir clases de excepción de acuerdo a las necesidades del invocador

Al usar API´s de terceros lo mejor es usar las expeciones para capturar la mayor cantidad de errores

Definir el flujo normal

SI se sigue lo anterior el código sera mas limpio, cuando hacemos código lo mejor es envolver las API externas para crear nuestras propias excepciones y crear un controlador para procesar los cálculos cancelados

No devolver Null

No debemos usar null en nuestro codigo, lo mejor sera usar las excepciones

No pasar Null

Nunca debemos usar o pasar null a un metodo a no ser que sea desde una API que lo requiera

Limites

Utilizar código de terceros

En el momento de usar API de terceros debemos usar solo las partes que necesitemos, es decir, si solo necesitamos que de una API se cambie una posición solo usaremos ese sin traer todo el API para el programa que estemos haciendo , si encontramos cosas que se puedan mejorar las mejoramos

Explorar y aprender limites

Al momento de usar código de terceros lo que debemos hacer es realizar pruebas para ver si nos sirve en nuestro programa en vez de leer la documentación, esto nos ahorra tiempo y esfuerzo

Aprender log4j

A veces los datos que parecen que no tienen importancia son el núcleo de un código

Las pruebas de aprendizaje son algo más que gratuitas

Las pruebas de aprendizaje son importantes para adquirir conocimientos ya que nos prueban a usar el código de terceros correctamente en nuestro código

Usar código que todavía no existe

En algún momento debemos crear nuestras propias API porque lo necesitamos para algo en especifico y no existe algo similar, lo mejor es superar el miedo de crearla

Limites limpios

Debemos tener control de nuestro código y no dejarlo en manos de terceros, esto va enfocado a tener un mínimo de puntos a tocar si se produce algún cambio en el limite