clean code

clean code es un libro de buenas practicas de programagacion escrito **Robert C. Martin**, el mismo inicia indicando que “si lees esto es por dos razones, eres programador y quiere ser mejor programador, que bueno, necesitamos mejores programadores”

primer capitulo, introduccion

en este libro se va a vewer mucho codigo, tanto codigo que al terminar con el libro vamos a saber diferencia buen codigo de mal codigo y el como transformar mal codigo en buen codigo

si bien hoy en dia se cuestione la continuidad de la figura del “programador” o “el codigo” debido a la constante busqueda de lenguajes que automaticen la programacion o que se enfoquen en ambitos especificos, o inclusive la aparicion de las IA, jamas desaparecera el codigo, ya que nadie puede mediante vagos sentimiento o expresiones redactar bien los requerimientos del programa (algo tan formal e importante como el codigo en si mismo), hasta que no se pueda crear un lenguaje que haga que la maquina entienda a la persona y no que la persona se de a entender a la maquina, vamos a seguir teniendo programadores

codigo malo

el autor se enoja con otro autor(Kent Beck) al mencionar en un libro propio que “el codigo bueno importar” era una fragil premisa, para Roberto era una premisa principal, el a visto empresas fundirse por mal codigo, muchas veces uno esta apurado, por tiempos de entrega por que tiene que correr a hacer otras partes del proyecto, porque se atraso, por lo que sea, sin embargo codigo que se hace mal atrasa mas que el retraso en si mismo ya que existe un fenomeno descrito como “wading” que consiste en ir vadeando en el codigo, esto se refiere a que el mal codigo en el intento de entenderlo atrasa mas que el tiempo de desarrollo, respecto de “dejarlo asi, despues lo corrijo” el autor hace alusion a la ley de Le’ Blanc: mas tarde equivale a nunca

el costo total de un desastre

si llevas programando el suficiente tiempo seguramente el codigo enquilombado de otro programador te a retrasado, el mal codigo retrasa, en una empresa se puede dar que el atraso lleve a que la misma contrate mas oprgramadores que terminen atrasando mas ya que estos saben menos del proyecto que los programadores ya existentes por tanto retrasan mas el proyecto

el gran rediseño

el autor parece relatar sus propias vivencias, habla de un famoso “gran rediseño” que aparece luego de que los desarrolladores infieran que el proyecto no puede continuar asi y demandan a la empresa que de luz verde a un rediseño que permita mejorar las condiciones del codigo viejo, el asunto con esto es que ahora la mtad de los developers estan trabajando en el sistema viejo y mal hecho y la otra mitad esta apurada en el rediseño haciendo que para cuando termina sea mas rentable hacerle a este rediseño un rediseño

attitude y the primal conundrum

aca menciona el como los messed code son en realidad nuestra culpa, las deadlines las pone el pm ahora el que estirarlas en post del buen codigo depende enteramente de nosotros,es prioridad en nosotros hacernos cargo de lo que nos toca, hacer buen codigo antes que cualquier otra cosa, la unica manera de ir rapido es haciendo buen codigo

el arte del codigo limpio

al igual que en el arte reconocer buen arte no significa saber hacer buen arte, lo mismo con el codigo, hacer buen codigo e identificar buen codigo no van de la mano

que es codigo limpio

aca el autor cita a varias eminencias de la programacion

Bjarne Stroustrup (inventor de c++)

“me gusta mi codigo elegante y eficiente, tiene como mojetivo que los bugs sean dificiles de ocultar, las dependencias faciles de mantener, el manejo de eerrores completado mediante a una articulada estrategia, y la performance cercana a la optima para no tentar a la gente a hacer codigo enquilombado para optimizaciones innecesarias, el codigo limpio hace una cosa”

Grady Booch (autor de Object Oriented Analysis and Design with Applications)

“el codigo limpio es simpe y directo, se lee como una prosa bien escrita, este nunca oscurese el intento de diseño sino que esta lleno de crispadas abstracciones y lineas de control directas”

Ward Cunningham (inventor de wiki, el motivador de patrones de diseño, lider en el desarrollo de smalltalk y el padrino de todos los que les importa el buen codigo)

“sabes que que estas trabajando en codigo limpio cuando cada rutina que lees es mas linda de lo que esperabas, podes decir que es codigo hermoso cuando el codigo te hace ver como si el lenguaje utilizado estaba hecho para el problema”

escuelas de pensamiento

y que piensa el tio bob (el mismo) sobre esto?, el autor hace un paralelismo con las artes marciales, existen varias escuelas con distintas tecnicas en cada una y hasta distinta filosofia, ahora el alumno puede centrarse en una y meujorar en la misma pero a la larga el mismo puede estudiar en distintqas escuelas aprnder distintas tecnicas, crear las suyas propias en inclusive crear su propia escuela, uno debe aprender de tantos autores como pueda ya que asi podra encontrar su propia escuela de como programar

nosotros somos los autores

somos programadores, aprendemos de otros programadores pero somo autores de nuestro propio codigo por tanto nuestras enseñanzas deben ser transmitidas tanto como la de los autores en quienes nos inspiramos

ley del boy scout

siemopre hay que dejar las cosas mejor que como las encontramos, sin vemos codigo malo tenemos que esfrozarnos en dejalro mejor que como lo encontramos

capitulo 2: nombres con significado

los nombres para la programacion son todo, todo tiene un nombre, las variables, los entornos, las funciones, las clases, los archivos.jar, etc

usar nombres que revelen intencionalidad

los nombres tanto de variables, funciones, clases deben de detallar el porque de su existencia, si hace falta aclarar con un comentario, no deberia de estar en un principio

un ejemplo:

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

para que es ese codigo? que hay en la lista, que es la x, que significa el 4, no dice absolutamente nada

ahora

```
public List<int[]> getFlaggedCells() {
List<int[]> flaggedCells = new ArrayList<int[]>();
for (int[] cell : gameBoard)
if (cell[STATUS_VALUE] == FLAGGED)
flaggedCells.add(cell);
return flaggedCells;
}
```

pasa a tener algo de centido, se entiende que es cada elemento y porque existe, asi y todo esto es hasta mejorable

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

utilizando clases por ejemplo y metodos de clase la cosa se vuelve hasta novelistica, se puede leer como si fuera un texto y no codigo, por eso es importante el elegir buenos nombre

evita la desinformacion

evite la desinformacion, abreviaturas como hp, aix y sco son nombres en el sistema unix, si por ejemplo tenes una variable que representa la hipotenusa y pensas que hp es un buen nombre, pues no, alguien que viene de unix (o que le gusta la mecanica como a mi) se podria confundir, si tu conjunto de cuentas se va a llamar *AccountList* mas te vale que sea una lista, ya que list en programacion tiene significado y en caso de no ser una lista podria llevar a confucion, para eso es mejor llamarlo *accountGroup*, *bunchOfAccounts* o mejor *accounts* es mejor que algo que te pierda.

mucho cuidado con usar nombres que varien muy poco entre si, alguien que no hizo el codigo y esta enfocado en otro modulo cuanto se tomara en entender que *XYZControllerForEfficientHandlingOfStrings* y *XYZControllerForEfficientStorageOfStrings* no son la misma variable

pronunciar cosas similares de manera similar es informacion, usar pronunciaciones inconsistente es desinformacion

hacer distinciones significativas

muchas veces se tiende a agregar palabras de ruido y/o numeros a una variable, clase, objeto para distinguirlo de otro muy similar, la preguntas es: si hacen falta maromas linguisticas o de caracteres para distinguirtlos, porque los distinguirias? que diferencia ProductData de ProductInfo? porque harias la distincion, cual deberia usar yo?, ninguna variable deberia llenvar variable en su nombre, mismo tabla, clase, etc. en que mundo nameString es mejor que name? harias un name que fuera un flotante?, el error de campo que muestra el autor para mostrar un caso real es el siguiente:

```
    getActiveAccount();
    getActiveAccounts();
    getActiveAccountInfo();
```

si yo estuviera programando en el proyecto, como se como funciona getActiveAccount(), por usaria ese en ves de getActiveAccountInfo().

usa nombre pronunciables

gran parte del cerebro humano esta abocado a entender y pronuncia el lenguaje, es un crimen no aprovechar esa gran parte teniendo en cuenta que programar es una actividad social, el autor cita una compañia que entre sus variables tenia una llamada: genymdhms, (generation date, year, month, day, hour, minute, and second), en ingles suena mas divertido, no es serio entrar a una reunion laborar al grito de “the “gen why emm dee aich emm ess” has a glitch”, comparemos 2 codigos distintos para entenderlo

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

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

imaginate teniendo una conversacion seria con tu jefe sobre ambos codigos, ahi entenderas el punto de nombrar cosas pronunciables

usar nombres buscables

los nombres de una sola letra y las constantes numericas tiene una particularidad y es que en un proyecto grande buscar un 7 o una e podria ser infinitamente mas dificil de buscar un “MAX_CLASSES_PER_STUDENT”, lo ideal es que nombre de una sola letra solo sean usados como variables locales en metodos cortes (i para un for por ejemplo), el largo de una variable debe corresponder con el alcance de la misma, una variable local puede ser corta ahora una global o el nombre de una clase debe ser tan extenso como su uso lo marque.

evitar el encodings

ya en la programacion hay mucho encodings como para andar agregando el propio, imaginate presentarle a un nuevo miebro del equipo que ya viene con sus conocimiento de encoding el encoding que usa tu empresa, no tiene sentido

notacion hungara

antes en lenguajes mas viejos era importante usar HN para distinguir el tipo de la variable (una letra para el tipo, otra para identificar la variable), hoy dia no solo es innecesario sino que perjudica el entendimiento del codigo

prefijos de miembro

antes tambien se usaba los prefijos para saber a que funcion pertencian las variables, hoy en dia los ides ya te marcan donde empieza y donde termina una funcion o clase, los prefijos son innesecarios

interfaces e implementaciones

volvamo a lo mencionado sobre usar variable como parte del nombre, para un abstract factory no tiene sentido poner IshapeFactory por ejemplo, la i es innecesaria, no asumo que los usuarios de mi clase necesiten saber que estan usando una interfaz, si es necesario encodear algo sobre mi abstrac factory prefiero encodera la implementacion antes que la interafaz

evita el mapeo mental

si tenes que hacerte un mapa mental para saber que esa variable y tiene un nombre en uso entonces usare otra entonces vas mal, programadores inteligentes recuerdan que nombres de clase ya usaron, programadores profecionales no necesitan usar la memoria

nombres de clases

un nombre de clase debe ser siempre un sustantivo, jamas se deberia usar un verbo

nombres de metodos

estos si deben ser un verb y deben ser explicativos, tiene que detallar que hace el metodo en cuestion

no seas tierno

si el nombre de la variable es muy ingenioso igual y solo lo entenderan quienes conocen al autor, o sea las joditas guardalas para la charla post trabajo, el codigo no debe contener chistes

elegir una palabra por concepto

a veces elegir entre fetch retrive y get para metodos equivalente en clases diferentes es confuso, para que no? los ides actuales tiene segmentado por clase que metodos conllevan por tanto es innecesario darle a un mismo concepto distinto metodos en distintas clases si hacen los mismo

no te pises

supongamos que tenemos un metodo add que se encarga consistentemente en todas las clases que se implementa de concatenar un valor, si ahora yo uso add en una clases especifica para por ejemplo añadir un solo elemento a una lista, no me estoy pisando? un concepto una palabra es util cuando ese concepto se mantiene

usar nombres del dominio solucion

quiene lean tu codigo seguramente sean programadores, asi que porque no usar terminos tecnicos para ciertos casos, un JobQueue es algo que todo programador entendera a la primera

usar nombres del dominio del problema

a veces no habra ingenieros en tu equipo por tanto usar nombres del dominio del problema podria simplificarles la existencia

agragar contexto significativo

suponiendo que en un metodo encontramos variablers como nombre, apellido, calle, numeroDeCasa, ciudad y estado entendemos que eso son valores de direccion, ahora si encontramos “estado” por si solo cualquiera que sepa de patrones de diseño podria apresurar conclusiones

no añadir contexto gratuito

procura que en un mismo modulo no se sobreespecifique el codigo o clase en cuestion pertenece a ese modulo, a menos caracteres en el nombre mucho mejor, los nombres tienen que tener el largo suficiente como para entenderlos

la ultima palabra

no hay que tener miedo en renombrar un metodo o clase si es necesario, a veces los cambios son para mejor, muchas veces un programador ni se acuerda los nomrbes solo usa las herramientas del IDE para llamar a la clase, por eso el “Nombre definitivo” siempre esta en el momento ideal para modificarlo en caso de que este traiga una mejoria

capitulo 3 funciones

de los primeros dias de la programacion la unica estructura que aa dia de hoy sobrevive son las funciones

vamos con el siguiente ejemplo

```
public static String testableHtml(
PageData pageData,
boolean includeSuiteSetup
) throws Exception {
WikiPage wikiPage = pageData.getWikiPage();
StringBuffer buffer = new StringBuffer();
if (pageData.hasAttribute("Test")) {
if (includeSuiteSetup) {
WikiPage suiteSetup =
PageCrawlerImpl.getInheritedPage(
SuiteResponder.SUITE_SETUP_NAME, wikiPage
);
if (suiteSetup != null) {
WikiPagePath pagePath =
suiteSetup.getPageCrawler().getFullPath(suiteSetup);
String pagePathName = PathParser.render(pagePath);
buffer.append("!include -setup .")
.append(pagePathName)
.append("\n");
}
}
WikiPage setup =
PageCrawlerImpl.getInheritedPage("SetUp", wikiPage);
if (setup != null) {
WikiPagePath setupPath =
wikiPage.getPageCrawler().getFullPath(setup);
String setupPathName = PathParser.render(setupPath);
buffer.append("!include -setup .")
.append(setupPathName)
.append("\n");
}
}
buffer.append(pageData.getContent());
if (pageData.hasAttribute("Test")) {
WikiPage teardown =
```

```
PageCrawlerImpl.getInheritedPage("TearDown", wikiPage);
if (teardown != null) {
WikiPagePath tearDownPath =
wikiPage.getPageCrawler().getFullPath(teardown);
String tearDownPathName = PathParser.render(tearDownPath);
buffer.append("\n")
.append("!include -teardown .")
.append(tearDownPathName)
.append("\n");
}
if (includeSuiteSetup) {
WikiPage suiteTeardown =
PageCrawlerImpl.getInheritedPage(
SuiteResponder.SUITE_TEARDOWN_NAME,
wikiPage
);
if (suiteTeardown != null) {
WikiPagePath pagePath =
suiteTeardown.getPageCrawler().getFullPath (suiteTeardown);
String pagePathName = PathParser.render(pagePath);
buffer.append("!include -teardown .")
.append(pagePathName)
.append("\n");
}
}
}
pageData.setContent(buffer.toString());
return pageData.getHtml();
}
```

esta como ilegible no? bueno ese es el punto, muchos if anidados, no se entiende donde empieza ni donde termina algo entonces, el siguiente codigo

```
public static String renderPageWithSetupsAndTeardowns(
PageData pageData, boolean isSuite
) throws Exception {
boolean isTestPage = pageData.hasAttribute("Test");
if (isTestPage) {
WikiPage testPage = pageData.getWikiPage();
StringBuffer newPageContent = new StringBuffer();
includeSetupPages(testPage, newPageContent, isSuite);
newPageContent.append(pageData.getContent());
includeTeardownPages(testPage, newPageContent, isSuite);
pageData.setContent(newPageContent.toString());
}
return pageData.getHtml();
}
```

capaz no sabes como funciona cada funcion pero al menos entendes que se supone que hace cada cosa, es por eso que las funciones existen

pequeñas, small, cortas

la primera regla es que una funcion debe ser corta, la segunda es que debe ser muy corta, tan corto como sea posible, si una funcion supera las 30 lineas posiblemente algo no este correcto

bloque e identacion

todo bloque de if, else y while deberia no ser mayor a una linea, tanto asi como para poder ser remplazado con una funcion, 3 bloques de identacion alcanzan y sobran en este contexto, la idea es no tener funciones anidadas

do one thing

una funcion debe hacer una cosa, la debe hacer bien, la debe hacer ella

secciones con funciones

un programa deberia poder ser dividido en secciones donde cada una sea una funcion

un nivel de abstraccion por funcion

sabes que estas segmentando mal cuando una funcion toca al mismo tiempo dos niveles de abstraccion distintas, por ejemplo no tiene sentido que una funcion toque al mismo tiempo un strign y una clase abstracta

leer de arriba para abajo (stepdown rule)

la idea es que de arriba para abajo se pueda ir leyendo cada funcion en orden de ejecucion de manera tal que al llegar a la ultima encontremos a la cual depende de todas las demas

switches

no usarlos, o sea si, pero reducirlo lo mas que se pueda con polimorfismo

nombres descriptivo

mismo que en el capitulo anterior una funcion debe tener un nombre corto que diga exactamente lo que hace

argumentos de una funcion

deben ser lo mas reducidos posible, de no se evitable entonces estos deben ser tan descriptivos como sea posible, las funciones pueden ser categoriazadas como niladic (sin argumentos), monadic (1), diadic(2), triadic(3) y poliadic

capitulo 4 comentarios

un buen comentario puede ser muy util para terceros para entender un codigo, el problema es precisamente que deben ser buenos, por eso voy a separar que es un buen comentario de lo que no, y es que muchas veces se usan comentarios para enmdendar algo que sencillamente es mal codigo

buenos comentarios

legales

a veces por estadares de la compañia uno debe poner comentarios legales

```
// Copyright (C) 2003,2004,2005 by Object Mentor, Inc. All rights reserved.
// Released under the terms of the GNU General Public License version 2 or
later.
```

informativos

a veces pueden dar de manera resumida informacion interesante

```
// format matched kk:mm:ss EEE, MMM dd, yyyy
Pattern timeMatcher = Pattern.compile(
"\\d*:\\d*:\\d* \\w*, \\w* \\d*, \\d*");
```

explica intencion

a veces esta bueno dar mas contexto sobre el porque de una decision

```
public void testConcurrentAddWidgets() throws Exception {
WidgetBuilder widgetBuilder =
new WidgetBuilder(new Class[]{BoldWidget.class});

String text = "'''bold text'''";
ParentWidget parent =
new BoldWidget(new MockWidgetRoot(), "'''bold text'''");
AtomicBoolean failFlag = new AtomicBoolean();
failFlag.set(false);
//This is our best attempt to get a race condition
//by creating large number of threads.
for (int i = 0; i < 25000; i++) {
WidgetBuilderThread widgetBuilderThread =
new WidgetBuilderThread(widgetBuilder, text, parent, failFlag);
Thread thread = new Thread(widgetBuilderThread);
thread.start();
}
assertEquals(false, failFlag.get());
}
```

clarificacion

si bien las variables y argumentos deberian hablar por si solos a veces o por librerias externas o porque el codigo no se deberia alterar es bueno aclarar cosas

```
Public void testCompareTo() throws Exception
{ WikiPagePath a = PathParser.parse("PageA");
WikiPagePath ab = PathParser.parse("PageA.PageB");
WikiPagePath b = PathParser.parse("PageB");
WikiPagePath aa = PathParser.parse("PageA.PageA");
WikiPagePath bb = PathParser.parse("PageB.PageB");
```

```
WikiPagePath ba = PathParser.parse("PageB.PageA");
assertTrue(a.compareTo(a) == 0); // a == a
assertTrue(a.compareTo(b) != 0); // a != b
assertTrue(ab.compareTo(ab) == 0); // ab == ab
assertTrue(a.compareTo(b) == -1); // a < b
assertTrue(aa.compareTo(ab) == -1); // aa < ab
assertTrue(ba.compareTo(bb) == -1); // ba < bb
assertTrue(b.compareTo(a) == 1); // b > a
assertTrue(ab.compareTo(aa) == 1); // ab > aa
assertTrue(bb.compareTo(ba) == 1); // bb > ba
}
```

no confuntir aclarar con hacer mal el codigo

advertir consecuencias

a veces es util para advertir de consecuencias a la hora de realizar una tarea

```
// Don't run unless you
// have some time to kill.
public void _testWithReallyBigFile()
{ writeLinesToFile(10000000);
response.setBody(testFile);
response.readyToSend(this);
String responseString = output.toString();
assertSubString("Content-Length: 1000000000", responseString);
assertTrue(bytesSent > 1000000000);
}
```

TODO

son los menos buenos de los buenos ya que muchas veces uno no termina haciendo lo que dice el TODO

```
//TODO-MdM these are not needed
// We expect this to go away when we do the checkout model
protected VersionInfo makeVersion() throws Exception
{ return null;
}
```

malos comentarios

practicamente todo lo demas, sobre especificar, comentarios redundantes (dicen algo que se ve en el codigo), comentarios que no se correlacionen con lo que esta en el codigo, comentarios tipo diario (flaco usa git), ruido (mas comentarios redundantes), usar comentarios para remplazar buenos nombres de variables o funciones, position markers, mucha informacion, codigo que no se usa, todo eso esta mal