Revisión de código

Rosa Victoria Villa Padilla

- La revisión del código es un estudio cuidadoso y sistemático del código fuente por parte de personas que no son el autor original del código. Todo programador lo tendrá que hacer en algún momento.

Propósitos de las revisiones

Propósitos de las revisiones

- **Mejorar el código.** Encontrar errores, anticipar posibles errores, verificar la claridad del código y verificar la coherencia con los estándares de estilo del proyecto.
- **Mejora del programador.** La revisión del código es una forma importante en que los programadores aprenden y se enseñan entre sí, sobre las nuevas características del lenguaje, los cambios en el diseño del proyecto o sus estándares de codificación y las nuevas técnicas. En proyectos de código abierto, particularmente, mucha conversación ocurre en el contexto de revisiones de código.

Propósitos de las revisiones

La revisión de código se practica ampliamente en proyectos de código abierto como **Apache** y **Mozilla**. https://blog.humphd.org/vocamus-1569/?p=1569

La revisión de códigos también se practica ampliamente en la industria. En **Google**, no puede insertar ningún código en el repositorio principal hasta que otro programador lo haya aprobado en una **revisión de código.**

Normas de estilo

- La mayoría de las empresas y los grandes proyectos tienen **estándares de estilo de codificación** (por ejemplo, Google Java Style http://google.github.io/styleguide/javaguide.html). Estos pueden ser bastante detallados, incluso hasta el punto de especificar espacios en blanco (la profundidad de la sangría) y dónde deben ir las llaves y los paréntesis. Este tipo de preguntas a menudo conducen a… (civil war), ya que terminan siendo una cuestión de gusto y estilo.

```java
public class MuteQuack implements QuackBehavior {

    public void quack() {
        System.out.println("<< Silence >>");
    }
}
```

```java
public class MuteQuack implements QuackBehavior
{
    public void quack()
    {
        System.out.println("<< Silence >>");
    }
}
```

Normas de estilo

Para Java, hay una guía de estilo general (desafortunadamente no actualizada para las últimas versiones de Java). Algunos de sus consejos son muy específicos:

- La declaración de una llave { debe estar al final de la línea que comienza la declaración compuesta; la declaracion de la llave } debe comenzar en una línea nueva y estar identado a la altura de la instrucción compuesta.

Normas de estilo

Sin embargo, es importante ser **autoconsistente**, y es muy importante **seguir las convenciones del proyecto** en el que está trabajando. Si eres el programador que reformatea cada módulo que tocas para que coincida con tu estilo personal, tus compañeros te odiarán, y con razón. Sé un jugador de equipo.

Ejemplo 01 de lo que no debes hacer

```
public static int dayOfYear(int month, int dayOfMonth, int year) {
    if (month == 2) {
        dayOfMonth += 31;
    } else if (month == 3) {
        dayOfMonth += 59;
    } else if (month == 4) {
        dayOfMonth += 90;
    } else if (month == 5) {
        dayOfMonth += 31 + 28 + 31 + 30;
    } else if (month == 6) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31;
    } else if (month == 7) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30;
    } else if (month == 8) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31;
    } else if (month == 9) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31;
    } else if (month == 10) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30;
    } else if (month == 11) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31;
    } else if (month == 12) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31 + 31;
    }
    return dayOfMonth;
}
```

Don’t Repeat Yourself
No repitas

- El código duplicado es un riesgo para la seguridad. Si tiene un código idéntico o muy similar en dos lugares, entonces el riesgo fundamental es que haya un error en ambas copias, y algunos programadores de mantenimiento corrigen el error en un lugar pero no en el otro.

Don’t Repeat Yourself
No repitas

Copiar y pegar es una herramienta de programación enormemente tentadora, pero peligrosa. Cuanto más largo sea el bloque que está copiando, más riesgoso será.

- **Don’t Repeat Yourself** , o DRY para abreviar, se ha convertido en el mantra de los programadores.

  El metodo dayOfYear está lleno de código idéntico. ¿Cómo lo corregirías usando DRY?
-
-

Comentarios

Comentarios donde se necesita

Los buenos desarrolladores de software escriben comentarios en su código, y lo hacen con criterio.

Los buenos comentarios deberían hacer que el código sea más **fácil de entender**, más **seguro contra errores** (porque se han documentado suposiciones importantes) y **listo para el cambio**.

Comentarios donde se necesita

Un tipo de comentario crucial es una especificación, que aparece sobre un método o sobre una clase y documenta el comportamiento del método o la clase. En Java, esto se escribe convencionalmente como un comentario Javadoc, lo que significa que comienza con **/**** e incluye @ -syntax, como **@param** y **@return** para los métodos.

$$f(n) = \begin{cases} n/2 & \text{if } n \equiv 0 \pmod 2 \\ 3n+1 & \text{if } n \equiv 1 \pmod 2. \end{cases}$$

```
/**
 * Compute the hailstone sequence.
 * See http://en.wikipedia.org/wiki/Collatz_conjecture#Statement_of_the_problem
 * @param n starting number of sequence; requires n > 0.
 * @return the hailstone sequence starting at n and ending with 1.
 *         For example, hailstone(3)=[3,10,5,16,8,4,2,1].
 */
public static List<Integer> hailstoneSequence(int n) {
    ...
}
```

Comentarios donde se necesita

Otro comentario crucial es uno que especifica la procedencia o la fuente de un fragmento de código que se copió o adaptó de otra parte.

```
// read a web page into a string
// see http://stackoverflow.com/questions/4328711/read-url-to-string-in-few-lines-of-java-code
String mitHomepage = new Scanner(new URL("http://www.mit.edu").openStream(), "UTF-8").useDelimiter("\\A").next();
```

- Una de las razones para documentar las fuentes es evitar violaciones de **derechos de autor**. Los pequeños fragmentos de código en Stack Overflow suelen ser de dominio público, pero el código copiado de otras fuentes puede ser propietario o estar cubierto por otros tipos de licencias de código abierto, que son más restrictivas.

  Otra razón para documentar las fuentes es que el código puede quedar **desactualizado**; La respuesta de Stack Overflow de la que proviene este código ha evolucionado significativamente en los años transcurridos desde su primera respuesta.

Comentarios Paja

Comentarios paja

- Algunos comentarios son malos e innecesarios. Las traducciones directas de código al inglés, por ejemplo, no hacen nada para mejorar la comprensión, porque debe asumir que su lector al menos conoce Java:

```
while (n != 1) { // test whether n is 1
    ++i; // increment i
    l.add(n); // add n to l
}
```

Comentarios en código oscuro

- El código oscuro debe obtener un comentario:

```
sendMessage("as you wish"); // this basically says "I love you"
```

También cuando los lenguajes te permite comprimir varias instrucciones en una sola, y simplemente no es fácil de deducir que es lo que esta haciendo esa linea “mágica”.

```
return "(%s %s)" % (self.dato(), map(lambda v: v.dato(), self.vecinos()))
```

Comentarios

- El metodo dayOfYear necesita algunos comentarios, ¿dónde los pondrías?

- Por ejemplo, ¿no documentaría si se month se ejecuta de 0 a 11 ó de 1 a 12?

Falla rápido

- Failing fast significa que el código debería revelar sus errores lo antes posible. Cuanto antes se observa un problema (cuanto más cercano a su causa), más fácil es encontrarlo y solucionarlo. La comprobación estática falla más rápido que la comprobación dinámica, y la comprobación dinámica falla más rápido que producir una respuesta incorrecta que puede dañar el cálculo posterior.
- El método dayOfYear no falla rápidamente: si le pasas los argumentos en el orden incorrecto, devolverá silenciosamente la respuesta incorrecta. De hecho, la forma en que está diseñado dayOfYear es muy probable que un no estadounidense apruebe los argumentos en el orden incorrecto.

Ejemplo 01 de lo que no debes hacer

```
public static int dayOfYear(int month, int dayOfMonth, int year) {
    if (month == 2) {
        dayOfMonth += 31;
    } else if (month == 3) {
        dayOfMonth += 59;
    } else if (month == 4) {
        dayOfMonth += 90;
    } else if (month == 5) {
        dayOfMonth += 31 + 28 + 31 + 30;
    } else if (month == 6) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31;
    } else if (month == 7) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30;
    } else if (month == 8) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31;
    } else if (month == 9) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31;
    } else if (month == 10) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30;
    } else if (month == 11) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31;
    } else if (month == 12) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31 + 31;
    }
    return dayOfMonth;
}
```

Evita los números mágicos

- En realidad, solo hay dos constantes que los científicos de computación reconocen como válidas en sí mismas: 0, 1 y tal vez 2. (De acuerdo, tres constantes).
- Todas las demás constantes se llaman mágicas porque aparecen de la nada sin ninguna explicación.
- Una forma de explicar un número es con un comentario, pero una forma mucho mejor es declarar el número como una constante con un buen nombre claro.

Evita los números mágicos

El método dayOfYear está lleno de números mágicos:

- Los meses 2, ..., 12 serían mucho más fácil de leer como FEBRUARY , ..., DECEMBER .
- Los días del mes 30, 31, 28 serían más legibles (y eliminarían el código duplicado) si estuvieran en una estructura de datos como una matriz, lista o mapa, por ejemplo MONTH_LENGTH[month] .

Evita los números mágicos

- Los misteriosos números 59 y 90 son ejemplos particularmente perniciosos de números mágicos. No solo están sin comentarios ni documentados, sino que son en realidad el resultado de un cálculo realizado manualmente por el programador. **No codifique** las **constantes que haya calculado a mano**. Java es mejor en aritmética que tú. Cálculos explícitos como 31 + 28 hacer que la procedencia de estos números misteriosos sea mucho más clara. MONTH_LENGTH[JANUARY] + MONTH_LENGTH[FEBRUARY] Sería aún más claro.

Un propósito para cada variable

- En el método dayOfYear, el parámetro dayOfMonth se reutiliza para calcular un valor muy diferente: el valor de retorno de la función, que no es el día del mes.
- No reutilizar parámetros, y no reutilizar variables. Las variables no son un recurso escaso en la programación. Usalos libremente, dén les buenos nombres y deje de usarlos cuando deje de necesitarlos. Confundirás a tu lector si una variable que solía significar una cosa de repente comienza a significar algo diferente unas pocas líneas hacia abajo.

Ejemplo 02 de lo que no debes hacer

```
public static boolean leap(int y) {
    String tmp = String.valueOf(y);
    if (tmp.charAt(2) == '1' || tmp.charAt(2) == '3' || tmp.charAt(2) == 5 || tmp.charAt(2) == '7' || tmp.charAt(2) == '9') {
        if (tmp.charAt(3)=='2'||tmp.charAt(3)=='6') return true; /*R1*/
        else
            return false; /*R2*/
    }else{
        if (tmp.charAt(2) == '0' && tmp.charAt(3) == '0') {
            return false; /*R3*/
        }
        if (tmp.charAt(3)=='0'||tmp.charAt(3)=='4'||tmp.charAt(3)=='8')return true; /*R4*/
    }
    return false; /*R5*/
}
```

Usa buenos nombres

Buenos nombre de métodos y variables deben ser largos y autodescriptivos. Los comentarios a menudo se pueden evitar por completo al hacer que el código sea más legible, con mejores nombres que describen los métodos y las variables.

Usa buenos nombres

Por ejemplo, puedes reescribir

```
int tmp = 86400;  // tmp is the number of seconds in a day (don't do this!)
```

como

```
int secondsPerDay = 86400;
```

Usa buenos nombres

En general, los nombres de variables como a, b, tmp, temp y data son terribles, síntomas de la pereza extrema del programador. Cada variable local es temporal, y cada variable es datos, por lo que esos nombres generalmente carecen de significado. Es mejor **usar un nombre más largo y descriptivo**, para que su código se lea claramente por sí mismo.

Usa buenos nombres

Siga las convenciones de nomenclatura léxica del lenguaje. En Python, las clases suelen estar en mayúsculas, las variables en minúsculas y words_are_separated_by_underscores. En Java:

- metodosAreNamedWithCamelCaseLikeThis
- variablesAreAlsoCamelCase
- CONSTANTES_ARE_IN_ALL_CAPS_WITH_UNDERSCORES
- ClassesAreCapitalized
- packages.are.lowercase.and.separated.by.dots

Usa buenos nombres

- Los nombres de los métodos suelen ser frases verbales, como **getDate** o **isUpperCase** , mientras que los nombres de las variables y las clases suelen ser frases nominales. Elija palabras cortas y sea conciso, pero evite las abreviaturas. Por ejemplo, message es más claro que msg , y word es mucho mejor que wd . Tenga en cuenta que muchos de sus compañeros de equipo en clase y en el mundo real no serán hablantes nativos de inglés o español, y las abreviaturas pueden ser aún más difíciles para los hablantes no nativos.

Anécdota del **mal** uso de buenos nombres

Carlos [redacted]

Lenguajes de Programación y sus Paradigmas

7.25 ex3

Recolectores de Basura

Recolector de basura de aviación:

Para este tipo de recolector de basura sería una buena opción tener como opción para el procedimiento un algoritmo de "marcado y compactación de memoria", de esta manera se garantiza el hecho de que no queden fragmentaciones de memoria. Pero a esto se le puede añadir un incremento de tiempo ya que se necesita que el recolector de basura pasa rápido para que se garantice el regreso lo más pronto posible al funcionamiento del software. No recomendaría un algoritmo de "marcado y barrido" porque deja huecos en memoria los cuales no podrán ser utilizados hasta tiempo después y se necesita garantizar la memoria completa dentro del programa. Tampoco recomendaría el algoritmo de "para y chingas a tu madre" ya que este parte la memoria en 2 partes, lo cual no nos garantiza un óptimo manejo en espacio y memoriam y es probable que se necesite mucha memoria para procesar este tipo de software.

Recolector de basura estandar:

Para este sugeriría un algoritmo de "detente y copia con semi-espacios", por el medio de que de man estandar no sabemos cuanta memoria se ocupará por lo que manejar la mitad, es lo mejor. También p este caso se podría agregar una técnica incremental por tiempo, para tratar de lograr que el tiempo [illegible] Para manejar un recolector estándar no

Usa los espacios en blanco para ayudar al lector

- Use sangría consistente. El ejemplo de leap es malo en esto. El ejemplo de dayOfYear es mucho mejor en ese aspecto. De hecho, dayOfYear alinea muy bien todos los números en columnas, lo que facilita la comparación y la comprobación para un lector humano. Eso es un gran uso del espacio en blanco.

Usa los espacios en blanco para ayudar al lector

- Nunca use caracteres de tabulación para la sangría, solo los caracteres de espacio. No estamos diciendo que nunca debe presionar la tecla Tab, solo que su editor nunca debe poner un carácter de tabulación en su archivo fuente en respuesta a su presión sobre la tecla Tab. El motivo de esta regla es que las diferentes herramientas tratan los caracteres de tabulación de manera diferente, a veces expandiéndolos a 4 espacios, a veces a 2 espacios, a veces a 8. Si ejecuta "git diff" en la línea de comandos, o si ve su código fuente en un editor diferente, entonces la sangría puede estar completamente arruinada. Solo usa espacios. Configure siempre su editor de programación para insertar caracteres de espacio cuando presione la tecla Tab.

Ejemplo 03 de lo que no debes hacer

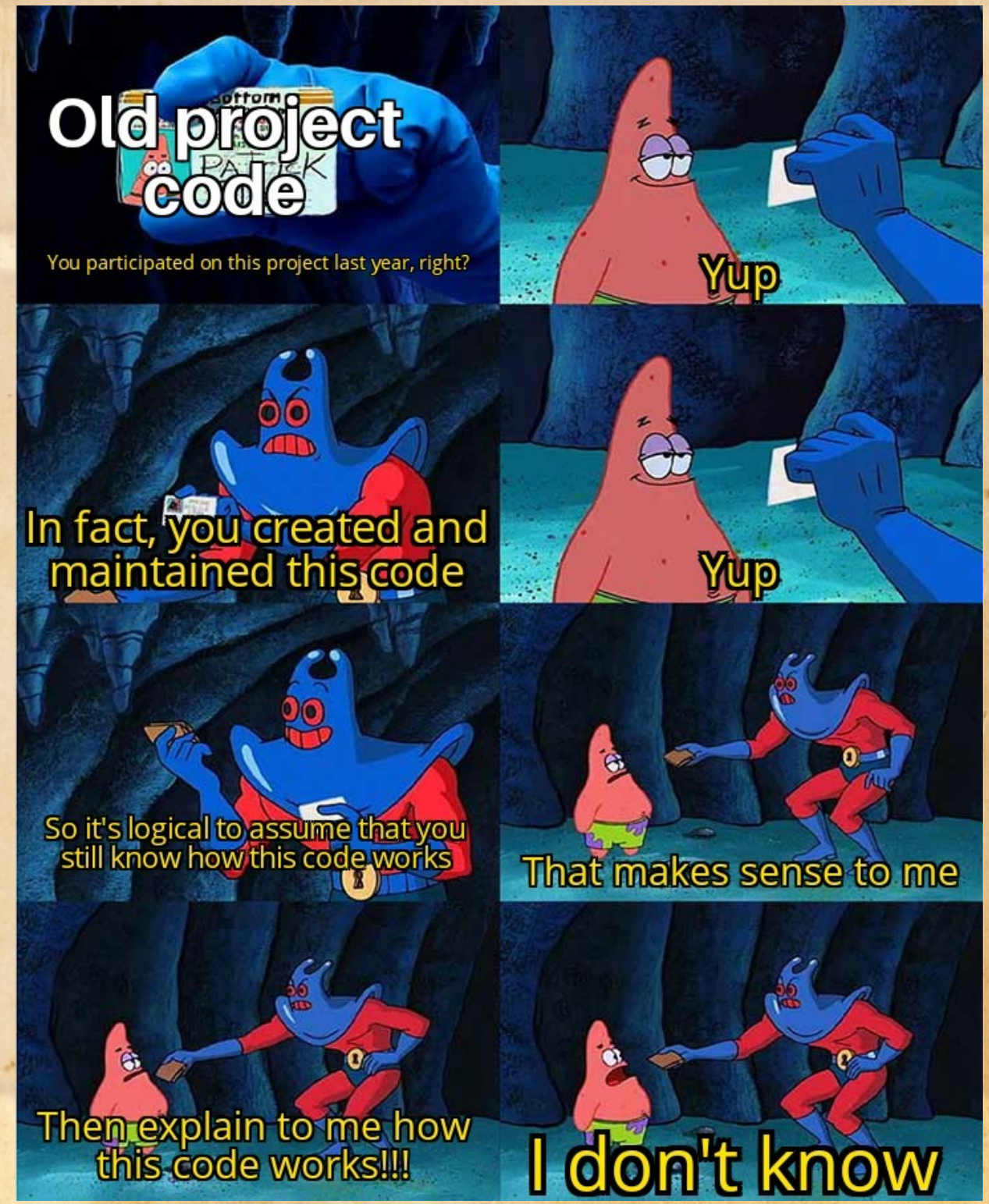
```
public static int LONG_WORD_LENGTH = 5;
public static String longestWord;

public static void countLongWords(List<String> words) {
   int n = 0;
   longestWord = "";
   for (String word: words) {
       if (word.length() > LONG_WORD_LENGTH) ++n;
       if (word.length() > longestWord.length()) longestWord = word;
   }
   System.out.println(n);
}
```

Los métodos deben devolver resultados, no imprimirlos

countLongWords no esta listo para el cambio. Se envía parte de su resultado a la consola, System.out. Eso significa que si desea usarlo en otro contexto, donde se necesita el número para algún otro propósito, como el cálculo en lugar de los ojos humanos, tendría que ser reescrito.

Los métodos deben devolver resultados, no imprimirlos

En general, solo las partes de mayor nivel de un programa deben interactuar con el usuario humano o la consola. Las partes de nivel inferior deben tomar su entrada como parámetros y devolver su salida como resultado. La única excepción aquí es la salida de depuración, que, por supuesto, se puede imprimir en la consola. Pero ese tipo de salida no debería ser una parte de su diseño, solo una parte de cómo depurar su diseño.


Old project code
You participated on this project last year, right?
Yup
In fact, you created and maintained this code
Yup
So it's logical to assume that you still know how this code works
That makes sense to me
Then explain to me how this code works!!!
I don't know