Clean Code

Joan Sebastián Ramírez Pérez

2016

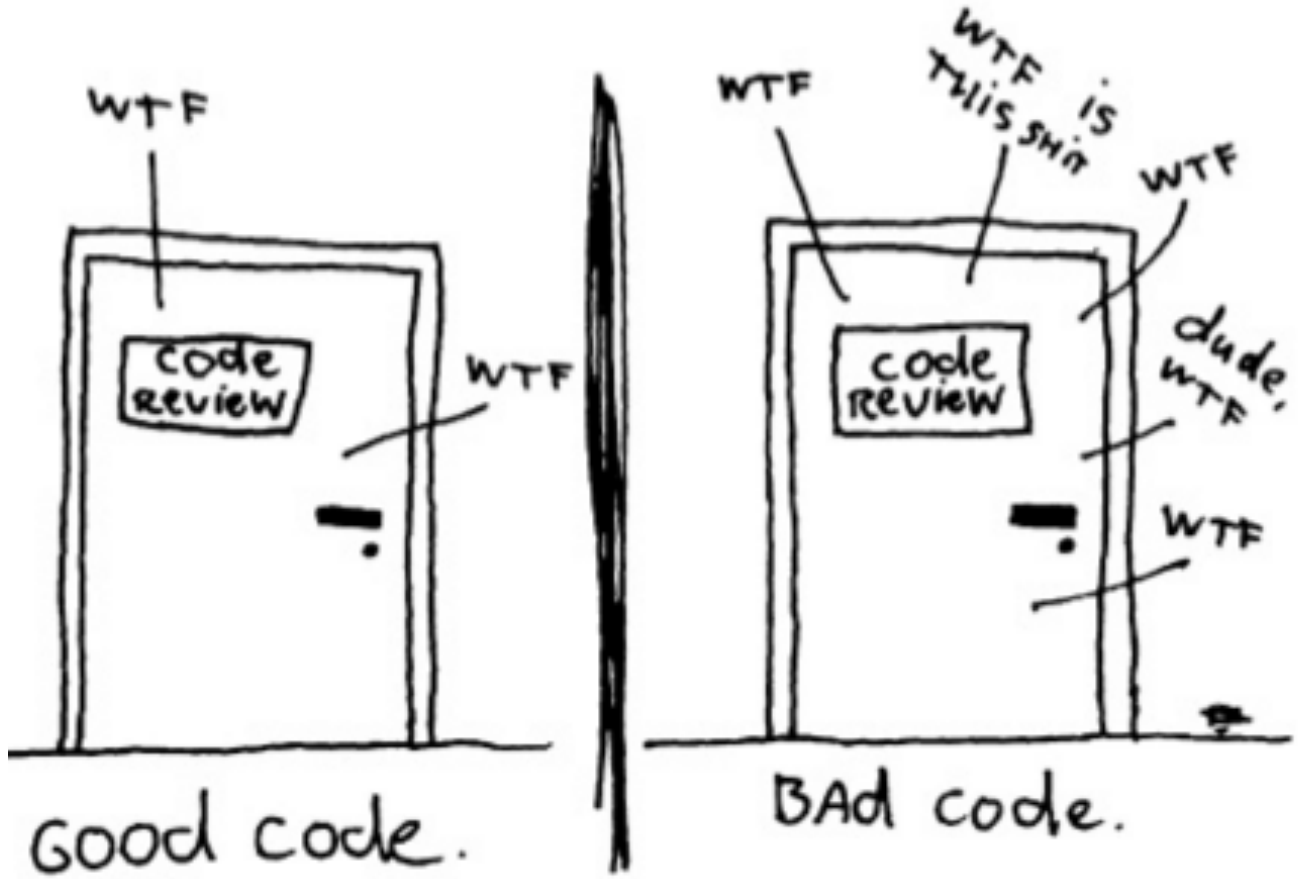
Agenda

- Código limpio
- Ejemplos
- Bibliografía

Agenda

- **Código limpio**
- Ejemplos
- Bibliografía

The ONLY VALID MEASUREMENT OF CODE QUALITY: WTFs/MINUTE



El coste total de un desastre

Si es programador desde hace dos o tres años, probablemente haya sufrido los desastres cometidos por otros en el código. Si tiene más experiencia, lo habrá sufrido en mayor medida. El grado de sufrimiento puede ser significativo. En un periodo de un año o dos, los equipos que avancen rápidamente al inicio de un proyecto pueden acabar a paso de tortuga. Cada cambio en el código afecta a dos o tres partes del mismo. Ningún cambio es trivial. Para ampliar o modificar el sistema es necesario comprender todos los detalles, efectos y consecuencias, para de ese modo poder añadir nuevos detalles, efectos y consecuencias. Con el tiempo, el desastre aumenta de tal modo que no se puede remediar. Es imposible.

Al aumentar este desastre, la productividad del equipo disminuye y acaba por desaparecer. Al reducirse la productividad, el director hace lo único que puede: ampliar la plantilla del proyecto con la esperanza de aumentar la productividad. Pero esa nueva plantilla no conoce el diseño del sistema. No conocen la diferencia entre un cambio adecuado al objetivo de diseño y otro que lo destroce. Por tanto, todos se encuentran sometidos a una gran presión para aumentar la productividad. Por ello, cometen más errores, aumenta el desastre y la productividad se acerca a cero cada vez más (véase la figura 1.1).

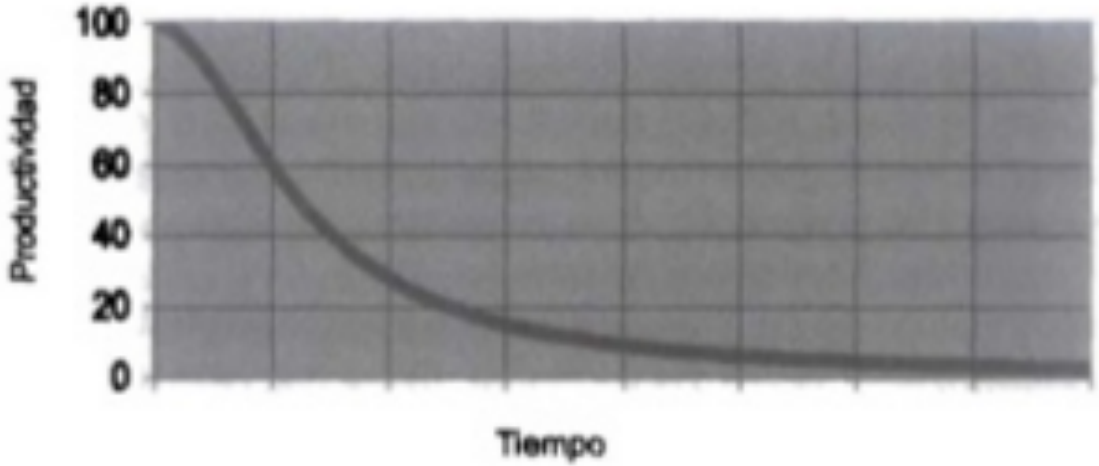
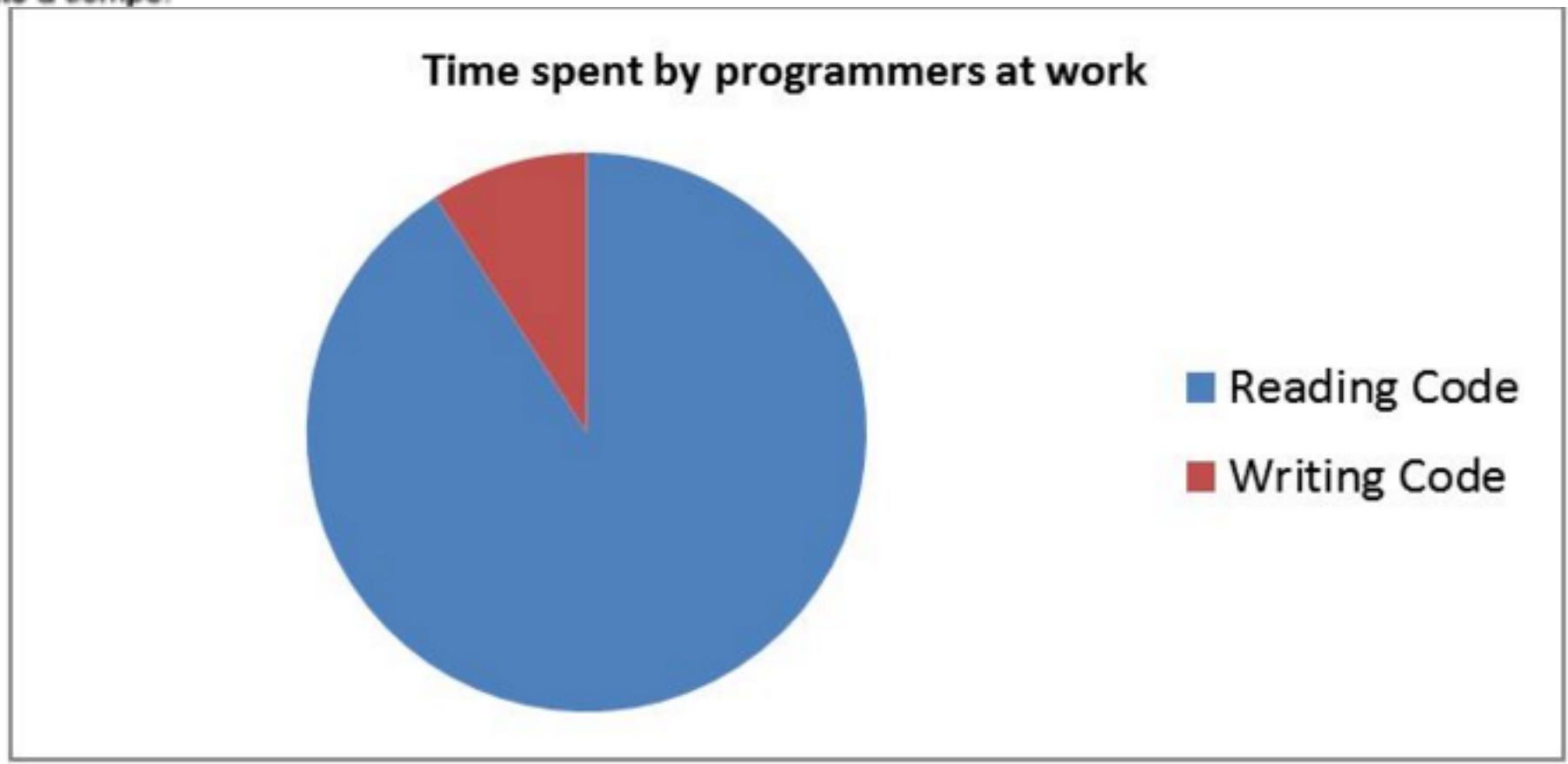


Figura 1.1. Productividad frente a tiempo.



Los comentarios no compensan el código incorrecto

Una de las principales motivaciones para crear comentarios es el código incorrecto. Creamos un módulo y sabemos que es confuso y está desorganizado. Sabemos que es un desastre y entonces decidimos comentarlo. Error. Mejor límpielo.

El código claro y expresivo sin apenas comentarios es muy superior al código enrevesado y complejo con multitud de comentarios. En lugar de perder tiempo escribiendo comentarios que expliquen el desastre cometido, dedíquelo a solucionarlo.

Explicarse en el código

En ocasiones, el código es un pobre vehículo de expresión. Desafortunadamente, muchos programadores lo entienden como que el código no es un buen medio de expresión. Esto es falso. ¿Qué prefiere ver? Esto:

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

o esto otro:

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

¿Qué es código limpio? (I)

- “La calidad de nuestro código puede ser vista desde distintos frentes. Uno de ellos tiene que ver con que el código esté lo más limpio de fallas posible, y otro tiene que ver con su estructura y mantenibilidad. En particular, la mantenibilidad ha sido la tortura mayor durante todos los decenios de la crisis del desarrollo de software desde mediados del siglo XX. Es por eso que mantener el código limpio no sólo es un tema de costos, sino que también es un tema de supervivencia profesional.”
  Bjarne Stroustrup, inventor de C++

¿Qué es código limpio? (II)

- “El código limpio es simple y directo, y se lee como prosa bien escrita que no obscurece las intenciones del diseñador, sino que esta lleno de abstracciones claras.” Grady Booch, creador de UML.
- “El código limpio parece estar hecho por alguien que le importa” Michael Feathers, autor de Working Effectively with Legacy Code.
- “Pasa todos los tests. No tiene duplicidades. Expresa las ideas de diseño del sistema. Minimiza el número de entidades como clases, métodos y similares” Ron Jeffries, autor de Extreme Programming Adventures in C#

¿Qué es código limpio? (III)

- “El código limpio puede ser leído y mejorado por un desarrollador distinto de su autor original. Tiene test unitarios y de aceptación. Tiene nombres con significado. Proporciona una forma de hacer las cosas en lugar de muchas alternativas.” Dave Thomas, Padrino de Eclipse.
- “Sabes que estás trabajando con código limpio cuando cada rutina que lees resulta ser como lo que esperabas encontrarte. Cuando parece que el lenguaje fue hecho para el problema que resuelve el código.” Ward Cunningham, inventor de la Wiki.

El Arte del código limpio

- La regla del Boy Scout.
- Nombres con sentido.
- Funciones.
- DRY (Don’t Repeat Yourself)
- Objetos y estructuras de datos.

La regla del Boy Scout

- “Deje el campamento más limpio de o que lo encontró.”
- No es sólo escribir bien el código, es mantenerlo limpio con el paso del tiempo.

Nombres con sentido (I)

Que revelan la intención y evitan la desinformación. Hacer distinciones significativas

- (por ejemplo no usar nombres como klass, a1 y a2)
- Pronunciables y buscables.
- Evite las notaciones de codificación (por ejemplo la Notación Hungara)
- Evite los mapas mentales
- Clases y métodos (Sustantivos para clases y objetos. Los métodos deben ser verbos o frases verbales) intención y evitan la desinformación.
- Que hagan distinciones significativas

Nombres con sentido (II)

- No seas simpático (nombre con sentido del humor no serán bien recibidos para quienes no comparten el sentido del humor del autor)
- Una palabra por concepto (ser coherente y usar la misma palabra a través de las clases, no sinónimos)
- No haga juegos de palabras
- Use nombres del dominio del problema y de la solución
- Contexto con significado
- No adicione contexto gratuito (no adicionar prefijos a los miembros de una clase ni en clases agregadas)

Funciones

- Pequeñas (No debería exceder las 20 líneas de código. Identado no mayor a 2)
- Un nivel de abstracción por función.
- Nombres descriptivos.
- Pocos argumentos (no usar flags como parámetros, se prefiere que sea monádica -un solo argumento-, evitar triadic-tres argumentos- y polyadic-más de tres argumentos-)
- Sin efecto colateral (efecto colateral genera acople, dependencias de orden, cambios inesperados de estado. Evitar argumentos de salida, usar el retorno)

- Separación comando-consulta (una función hace algo o responde a algo, no ambas cosas.)
- Prefiera la excepción al retorno de códigos.
- DRY (Don´t repeat yourself)
- Programación estructurada (Edsger Dijkstra “Toda función y cada bloque dentro de una función, deben ser una sola entrada y una sola salida.”. Es decir: un solo return por función, no breaks ni continues en un ciclo y nunca usar goto –estas reglas sirven poco en funciones pequeñas excepto lo del goto-)

DRY- Don’t repeat yourself

- Evitar la repetición en todas sus posibilidades:
- No repetir código: funciones, métodos, clases, etc. → Reutilizar.
- No repetir librerías.
- No repetir documentación.

Objetos y estructuras de datos

- Abstracción de datos
- Anti-simetría Dato/Objeto
- Ley de Demeter

Abstracción de datos

- Al esconder los datos, sólo queda funciones que representan el contrato con una clase.
- Esconder la implementación permite que sólo se piense en el qué. Evitando pensar en el cómo.
- Lo importante es expresar los datos en términos abstractos, y no pensar en términos de acceso a los datos con getter y setter.

Ley de Demeter

Heurística en la cual se define que un modulo no debe conocer los detalles internos de los objetos que manipula.

Un método f de una clase C solo debe llamar:

- Los métodos de C.
- Los métodos de un objeto creado por f.
- Los métodos de un objeto pasado como un argumento a f.
- Los métodos de un objeto mantenido en una variable instancia de C.

Olores y heurísticas

- Comentarios
- Ambiente
- Funciones
- General
- Nombres

Comentarios

C1: Información Inapropiada

C2: Comentario Obsoleto

C3: Comentario Redundante

C4: Comentario mal escrito

C5: Código comentado (zombie code)

Ambiente

E1: La generación requiere más de un paso

E2: Las pruebas requieren más de un paso

Funciones

F1: Demasiados argumentos

F2: Argumentos de salida

F3: Argumentos de indicador (banderas)

F4: Función muerta

General

G1: Varios lenguajes en un archivo de código

G2: Comportamiento evidentemente no implementado

G3: Comportamiento incorrecto en los límites

G4: Medidas de seguridad canceladas

G5: Duplicación

G6: Código en un nivel de abstracción incorrecto.

G7: Clases base que dependen de sus variantes

G8: Exceso de información

G9: Código muerto

G10: Separación vertical

General

G11: Incoherencia

G12: Desorden

G13: Conexiones artificiales (acoplamiento artificial)

G14: Envidia de las características

G15: Argumentos de selector

G16: Intención desconocida

G17: Responsabilidad desubicada

G18: Elementos estáticos incorrectos

G19: Usar variables explicativas

G20: Los nombres de la función deben indicar lo que hacen

General

G21: Comprender algoritmo

G22: Convertir dependencias lógicas en físicas

G23: Polimorfismo antes de If/Else o Switch/Case

G24: Seguir convenciones estándares

G25: Sustituir números mágicos por constantes con nombre

G26: Precisión

G27: Estructura sobre convención

G28: Encapsular condicionales

G29: Evitar condicionales negativas

G30: Las funciones solo deben hacer una cosa

General

G31: Conexiones temporales ocultas

G32: Evitar la arbitrariedad

G33: Encapsular las condiciones de límite

G34: Las funciones solo deben descender un nivel de abstracción

G35: Mantener los datos configurabas en los niveles superiores

G36: Evitar desplazamientos transitivos

Nombres

N1: Escoger nombres descriptivos
N2: Elegir nombres en el nivel correcto de abstracción
N3: Usar nomenclatura estándar siempre que sea posible
N4: Nombres inequívocos (no ambiguos)
N5: Usar nombres extensos para ámbitos extensos
N6: Evitar encodings
N7: Los nombres deben describir los efectos secundarios

Pruebas

T1: Pruebas insuficientes

T2: Usar una herramienta de cobertura de pruebas

T3: No ignorar pruebas triviales

T4: Una prueba ignorada es una pregunta sobre ambigüedad

T5: Probar condiciones límite

T6: Probar de forma exhaustiva junto a los errores

T7: Los patrones de fallo son reveladores

T8: Los patrones de cobertura de pruebas pueden ser reveladores

T9: Las pruebas deben ser rápidas

Pruebas

T6: Exhaustively Test Near Bugs Bugs tend to congregate. When you find a bug in a function, it is wise to do an exhaustive test of that function. You'll probably find that the bug was not alone.

T7: Patterns of Failure Are Revealing

T8: Test Coverage Patterns Can Be Revealing

T9: Tests Should Be Fast

Agenda

- Código limpio
- **Ejemplos**
- Bibliografía

¿Lo harías así?

```
private static List readLines(String fileName) {
  String line;
  ArrayList file = new ArrayList();
  try {
    BufferedReader in = new BufferedReader(new FileReader(fileName));
    while ((line = in.readLine()) != null)
      file.add(line);
    in.close();
  } catch (Exception e){
    System.out.println(e);
    return null;
  }
  return file;
}
```

¿Lo harías así?

```
25  //with side effects
26  class SessionService1 {
27
28      public Session login(User user, String password) {
29          if (user.getSession() != null) {
30              return user.getSession();
31          } else if (user.passwordMatched(password)) {
32              Session session = new Session();
33              user.setSession(session);
34              return session;
35          }
36          return null;
37      }
38  }
```

¿Lo harías así?

```
private static List readLines(String fileName) throws IOException {
  String line;
  List lines = new ArrayList();

  BufferedReader in = new BufferedReader(new FileReader(fileName));
  while ((line = in.readLine()) != null)
    lines.add(line);
  in.close();

  return lines;
}
```

¿ Cómo lo harías?

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

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

¿ Cómo lo harías?

```
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";
   /* ... */
};
```

¿ Cómo lo harías?

```
public class Part {
    private String m_dsc; // La descripción textual
    void setName(String name) {
        m_dsc = name;
    }
}
```

```
public class Part {
    String description;
    void setDescription(String description) {
        this.description = description;
    }
}
```

¿ Cómo lo harías?

```
1  int d;
2  // elapsed time in days
3  int ds;
4  int dsm;
5  int faid;
```

```
1  int elapsedTimeInDays;
2  int daysSinceCreation;
3  int daysSinceModification;
4  int fileAgeInDays;
```

¿ Cómo lo harías?

```
1  Customer[] customerList;
2  Table theTable;
```

```
1  Customer[] customers;
2  Table customers;
```

¿ Cómo lo harías?

```
1  var theCustomersListWithAllCustomersIncludedWithoutFilter;
2  var list;
```

```
1  var allCustomers;
2  var customersInOrder;
```

¿ Cómo lo harías?

```
1  public class EntitiesRelation
2  {
3  Entity o1;
4  Entity o2;
5  }
```

```
1  public class ProductWithCategory
2  {
3  Entity product;
4  Entity category;
5  }
```

¿Cómo lo harías?

```
1  string addressCity;
2  string addressHomeNumber;
3  string addressPostCode;
```

```
1  class Address
2  {
3  string city;
4  string homeNumber;
5  string postCode;
6  }
```

No basta con que el código funcione. El código que funciona suele ser incorrecto. Los programadores que se conforman con código funcional no se comportan de forma profesional. Puede que teman que no tienen tiempo para mejorar la estructura y el diseño del código, pero discrepo. No hay nada que afecte más negativamente a un proyecto de desarrollo que el código incorrecto. Los plazos incorrectos se pueden rehacer y los requisitos equivocados se pueden volver a definir. La dinámica incorrecta de un equipo se puede reparar pero el código incorrecto se corrompe y se convierte en una carga que arrastra al equipo completo. He visto equipos dominados por el desastre que han generado y que han dominado su destino.

Evidentemente, el código incorrecto se puede limpiar pero resulta muy costoso. Cuando el código se corrompe los módulos se insinúan unos a otros y generan multitud de dependencias ocultas y entrelazadas. La localización y división de dependencias antiguas es una tarea larga y complicada. Por otra parte, resulta relativamente sencillo mantener código limpio. Si comete un error en un módulo, es más fácil limpiarlo directamente. Mejor todavía, si cometió un error hace cinco minutos, es muy fácil limpiarlo ahora.

Por tanto, la solución consiste en mantener el código limpio y sencillo siempre que se pueda y no dejar que llegue a corromperse.

Agenda

- Código limpio
- Ejemplos
- **Bibliografía**

Bibliografía

- MARTIN, Robert. Código Limpio. Manual de estilo para el desarrollo ágil de software. Anaya Multimedia, 2012.