


RESUMEN Clean Code A Handbook of Agile Software Craftsmanship

Cheryll St

Related papers

[Download a PDF Pack](#) of the best related papers 



[Universidad Nacional de Salta](#)

Lorena Garcia

[Resumen de java](#)

Pascual Calzada

[Programacion Orientada a Objetos Con Java](#)

Ely LeoOnzita

RESUMEN

Clean Code A Handbook of Agile Software Craftsmanship

Autor: Robert C. Martin

Libro resumido por: Cheryll Stonestreet

CAPITULO 1:

Se muestra como crear código limpio, haciendo uso disciplinado de varias técnicas para obtener un código elegante. Una de las reglas en las que se basará Robert es en la analogía del Boy Scout, aplicado a la profesión de la informática: "Dejar el campamento más limpio de lo que se ha encontrado". Insistirá en este capítulo y los siguientes una y otra vez, que no nos debemos conformar con un código que funcione bien sino que también procuraremos dejarlo más limpio y legible que cuando lo encontramos.

Algunas de las citas mencionadas en el libro son las de Grady "el código limpio es simple y directo... y se lee como un texto bien escrito", otra la de Dave Thomas " el código limpio... Tiene pruebas de unidad y de aceptación; ofrece una y no varias formas de hacer algo" y Ron Jeffries dice que "el código simple...No contiene duplicados...minimiza el número de entidades como clases, métodos, funciones y similares." Una de las cosas que también aconseja Ron es crear un método o una clase más abstracta cuando tengo que hacer búsquedas de un elemento concreto en una colección, para poder disfrutar de modificar la implementación siempre que se desee o avanzar rápidamente al tiempo que conservo la posibilidad de realizar cambios posteriores. En conclusión, nada de duplicados, un objetivo, expresividad y pequeñas abstracciones.

CAPITULO 2:

Se recomienda elegir una palabra por concepto abstracto y mantenerla, según el tío Bob (como se llama a sí mismo), porque aunque los entornos de edición como Eclipse ofrecen la lista de métodos que puedes invocar para un determinado objeto no incluirá los comentarios de nombres de funciones y listas de parámetros. Por otro lado, **es confuso tener un controlador, un administrador y un control en la misma base de código**. Se debe separar los objetos en clases distintas.

Con los juegos de palabras tampoco es recomendable trabajar es decir, usar una misma palabra para fines distintos. Sólo usar así si los valores devueltos son semánticamente equivalentes. Nuestro objetivo siempre será facilitar la comprensión del código.

Debemos usar nombres de dominio de problemas, que significa que cuando no exista un término de programación para lo que esté haciendo, usar el nombre de dominio de problemas. Al menos el programador que mantenga su código podrá preguntar el significado a un experto en dominios.

Separar los conceptos de dominio de soluciones y de problemas es parte del trabajo de un buen programador y diseñador. El código que tenga más relación con los conceptos del dominio de problemas tendrá nombres extraídos de dicho dominio y los prefijos se usarán como último recurso.

CAPITULO 3:

Las funciones deben tener una longitud de aproximadamente 20 líneas.

Con relación a los bloques y sangrados se dice que las instrucciones tales como **if** y similares deben tener una línea de longitud. Las funciones no deben tener un tamaño excesivo que albergue estructuras anidadas. Por tanto, el nivel de sangrado de una función no debe ser mayor de uno o dos, para que sean más fáciles de leer y entender.

Un consejo que según Bob lleva más de 30 años de vigencia es el siguiente: "LAS FUNCIONES SOLO DEBEN HACER UNA COSA. DEBEN HACERLO BIEN Y DEBE SER LO ÚNICO QUE HAGAN".

Por otro lado, debe haber un nivel de abstracción por función es decir, que **las instrucciones de la función se deben encontrar en el mismo nivel de abstracción** porque de lo contrario los lectores no sabrán si una expresión es un concepto esencial o un detalle.

Todo debe ser leído descendentemente; como si fuera un conjunto de párrafos TO.

Instrucciones Switch: Por su naturaleza las instrucciones switch hacen N cosas. Cuando las usemos debemos asegurarnos de incluirlas en una clase de nivel inferior y no repetirlas. Para ello, se recurrirá al polimorfismo. Hay que **procurar cumplir el principio de responsabilidad única (SRP), también debe cumplir el principio de Abierto/Cerrado (OPC).**

Una solución para abordar el problema de la instrucción switch y su gran tamaño es ocultarla en una factoría abstracta e impedir que nadie la vea. La factoría usará la instrucción SWITCH para crear las instancias adecuadas de los derivados de la clase y las distintas funciones serán entregadas de forma polimórfica a través de una interfaz con el mismo nombre de la clase. **Aunque no siempre se podrá cumplir esta regla veremos el siguiente ejemplo:**

Listado 3.5. Employee y Factory.

```
public abstract class Employee {
    public abstract boolean isPayday();
    public abstract Money calculatePay();
    public abstract void deliverPay(Money pay);
}

-----

public interface EmployeeFactory {
    public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType;
}

-----

public class EmployeeFactoryImpl implements EmployeeFactory {
    public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType {
        switch (r.type) {
            case COMMISSIONED:
                return new CommissionedEmployee(r);
            case HOURLY:
                return new HourlyEmployee(r);
            case SALARIED:
                return new SalariedEmployee(r);
            default:
                throw new InvalidEmployeeType(r.type);
        }
    }
}
```

Usar nombres descriptivos: Siempre recordar el principio de Ward : "Sabemos que trabajamos con código limpio cuando cada rutina es más o menos lo que esperábamos". Un nombre descriptivo extenso es siempre mejor que uno breve y enigmático. Sea coherente con los nombres. Use las mismas frases,

sustantivos y verbos en los nombres de función que elija para los módulos.

Argumentos de funciones: El número ideal de argumentos para una función es cero. Siempre que sea posible debemos evitar el uso de tres argumentos en la función. Es mejor que como argumento pasemos una variable de instancia que un argumento normal para evitar que tengan que interpretarlo cada vez que lo vean.

A la hora de hacer pruebas es muy complicado hacerlas con argumentos, es mejor que la función no tenga argumentos. Un argumento de salida es la mejor opción, después de la ausencia de argumentos.

Formas monádicas habituales: Cuando el usuario ve una función con un argumento esperan que ese argumento se use para:

1. ser procesado, convertido a otra cosa y devuelto
2. o que se haga una pregunta sobre dicho argumento
3. otra cosa menos habitual pero que también se hace porque es muy útil es cuando el programa interpreta la función como un evento y usar **dicho argumento** para alterar el estado del sistema.

Argumentos de indicador: Pasar un valor Booleano a una función es una práctica totalmente desaconsejable, porque ya indica que hará dos cosas. (si true=esto/si false=aquello).

Objeto de Argumento: Cuando se pasan grupos de variables de forma conjunta, como x e y, es probable que formen parte de un concepto que se merece un nombre propio.

Lista de Argumentos: Si los argumentos variables se procesan de la misma forma, son equivalentes a un único argumento de tipo List.

Verbos y palabras clave: En formato monádico, la función y el argumento deben formar parte de un par (verbo(sustantivo)).

Sin efectos Secundarios: No hacer funciones que tengan efectos secundarios es decir, que hagan cosas ocultas(+ de dos cosas) porque provocará extrañas combinaciones temporales y dependencias de orden.

Argumentos de Salida: Debe evitarse las pausas cognitivas. Antes de la programación orientada a objetos, era necesario tener argumentos de salida. Pero con POO debe evitarse los argumentos de salida.

Separación de consultas de comando: Las funciones deben hacer algo o responder a algo, no ambas cosas porque causa confusión. Si vas a consultar y establecer al mismo tiempo es mejor separarlo, para evitar ambigüedades.

Mejor es devolver excepciones que códigos de error: Si usamos excepciones en lugar de códigos de error, el código de procesamiento del error se puede separar del código de ruta y se puede simplificar.

Try/Catch: Cuando usemos este tipo de bloques será más conveniente extraer el cuerpo de los bloques try y catch en funciones individuales.

```

public void delete(Page page) {
    try {
        deletePageAndAllReferences(page);
    }
    catch (Exception e) {
        logError(e);
    }
}

private void deletePageAndAllReferences(Page page) throws Exception {
    deletePage(page);
    registry.deleteReference(page.name);
    configKeys.deleteKey(page.name.makeKey());
}

private void logError(Exception e) {
    logger.log(e.getMessage());
}

```

Después de los bloques catch/finally no debe haber nada más.

El imán de dependencias Error.java: Este tipo de clases no es recomendable usarlas, porque son un imán para las dependencias. Cuando quieres añadir nuevos errores te lo planteas porque significa volver a implementarlo todo. **Por lo tanto es mejor usar excepciones en lugar de códigos de error.**

No Duplicados: Porque multiplican el riesgo de errores.

Programación Estructurada: No será necesario aplicar el principio de Edsger Dijkstra, porque siempre procuraremos que las funciones sean de reducido tamaño. Por tanto si podemos usar: **return, break y continue.**

Conclusiones:

- Recordar que las funciones son verbos y las clases los sustantivos.
- Que los programadores experimentados ven los sistemas como historias que contar, no como en programas que escribir.

CAPITULO 4:

Los Comentarios: Sólo deben usarse aquellos que estén actualizados, relevantes y precisos. Porque por norma general no debemos expresar nuestro código con comentarios, el código que escribimos debe ser capaz de expresarse sólo. Hay que recordar que el comentario realmente bueno, es aquel que no tenemos que escribir.

//TODO, se podrán usar este tipo de comentarios como recordatorios para corregir una implementación incorrecta en el futuro es decir, **tareas que el programador piensa que debería haber hecho pero no así.** Pero es necesario corregir y quitarlos lo antes posible.

Cuando hacemos Amplificación, que es amplificar la importancia de algo relevante se podrá usar comentarios.

Si uso **una API pública,** debo crear javadoc de calidad para la misma.

Los comentarios de cambios Periódicos deben ser eliminados, para eso existen los sistemas de control de código fuente como SVN.

No se deberá usar comentarios si se puede usar una función o una variable.

Se debe eliminar marcadores de posición innecesarios. Ej: Mi famoso /***/**

También, se debe eliminar el código comentado, como también los comentarios HTML, porque son una aberración según Bob. Si finalmente es necesario usar un comentario **la conexión entre el comentario y el código** que describe deberá ser evidente. Para código **no dirigido a consumo público** no deberá usarse javadoc.

CAPÍTULO 5:

Cuando codifiquemos debemos dejar que los demás aprecien que se trata de un código programado por un profesional, no una masa amorfa de código. El **formato** de código es importante, porque se basa en la comunicación. Debo recordar que la **principal preocupación de un programador no es que su código funcione, sino que a pesar de que cambie las versiones la legibilidad del código se mantenga.**

La legibilidad establecerá los precedentes que afectan a la capacidad de mantenimiento y ampliación mucho después que el código cambie. El estilo del programador y su disciplina sobrevivirán, aunque el código no lo haga.

La longitud máxima recomendada en un archivo es 500 líneas y mínimas pueden ser 200. Un archivo de código debe ser como un artículo de periódico. Los elementos superiores del archivo deben proporcionar conceptos y algoritmos de nivel superior. Los detalles deben ir aumentando a medida que avanzamos, hasta que en la parte final encontremos las funciones de nivel inferior del archivo. También **deberemos cuidar el interlineado entre un concepto y otro. En cuanto a densidad vertical** si la apertura separa los conceptos, la densidad vertical implicará asociaciones.

Distancia Vertical: No se debe hacer al usuario saltar de una función a otra para adivinar la relación y funcionamiento de las funciones. **Los conceptos relacionados entre sí deben mantenerse juntos verticalmente. Esta regla no funcionará con conceptos de archivos independientes. Por lo tanto no se deberá separar conceptos relacionados en archivos independientes, a menos que haya un motivo de peso. Por la misma razón debe evitarse usar las variables protegidas.**

Variables de Instancia: Deben ser declaradas en la parte superior de la clase, no deben aumentar la distancia vertical de las variables, ya que en una clase bien diseñado se usan en casi todos los métodos.

Funciones dependientes: Si una función invoca a otra, deben estar verticalmente próximas, y la función de invocación debe estar por encima de la invocada siempre que sea posible.

El concepto de afinidad: No solo incluye afinidad de conceptos sino que también puede generarse por un grupo de funciones que realizan una operación similar.

Orden Vertical: Las dependencias de invocaciones de funciones debe apuntar hacia abajo o sea, que la función invocada debe estar por debajo de la que invoca. **Esperamos que los conceptos más importantes aparezcan antes y que se expresen con la menor cantidad de detalles sobrantes.**

Formato Horizontal: En torno a 45 caracteres. Evidentemente los programadores prefieren líneas menos largas. Prohibido poner más de 120 caracteres en una misma línea.

Apertura y densidad horizontal: Cuando hagamos asignaciones separe con tabulador el signo de = de las variables. Cuando se trate de ecuaciones los términos se separarán mediante espacios, para saber quien

tiene mayor o menor precedencia.

Romper el Sangrado: Cuando se trata de instrucciones breves sucumbimos a la tentación de romper la regla del sangrado aunque no es recomendable. Es mejor mantenerlo.

Reglas de Equipo: Los equipos de programadores acordarán el único estilo de formato y cada integrante deberá aplicarlo.

CAPITULO 6:

Objetos y Estructura de Datos: Las variables como normal general deben ser privadas, para que nadie dependa de ellas, para poder cambiar su implementación cuando lo deseemos.

No queremos mostrar los detalles de los datos, sino expresarlos en términos abstractos. Esto no se consigue simplemente mediante interfaces o métodos de establecimiento y recuperación. Hay que meditar seriamente la forma óptima de representar los datos que contiene un objeto. La peor opción es añadir métodos de establecimiento y recuperación a ciegas.

Asimetría de Datos y Objetos: La diferencia entre objetos y estructuras de datos está en que los **Objetos ocultan sus datos tras abstracciones** y muestran funciones que operan en dichos datos. **La estructura de datos muestra sus datos** y carece de funciones con significado.

Método Polimórficos son efectivos para la orientación de objetos. Donde por ejemplo una misma función puede retornar el área de varias figuras diferentes.

La dicotomía fundamental entre objetos y estructura de datos está en la siguiente declaración de Bob:

El código por procedimientos (el que usa estructuras de datos) facilita la inclusión de nuevas funciones sin modificar las estructuras de datos existentes. El código orientado a objetos, por su parte, facilita la inclusión de nuevas clases sin cambiar las funciones existentes.

El complemento también es cierto:

El código por procedimientos dificulta la inclusión de nuevas estructuras de datos ya que es necesario cambiar todas las funciones. El código orientado a objetos dificulta la inclusión de nuevas funciones ya que es necesario cambiar todas las clases.

Los programadores experimentados saben que la idea de que todo es un objeto es un mito. En ocasiones solamente queremos sencillas estructuras de datos con procedimientos que operen en las mismas.

Choque de trenes: Conviene dividir las invocaciones para no incumplir la ley de Demeter. Las funciones no deben saber demasiado, no deben saber desplazarse por numerosos objetos diferentes. **Para cumplir la ley de Demeter** se debe ocultar los detalles internos de los objetos (**ocultar sus datos y mostrar sus operaciones**), y si es una estructura de datos debe mostrar su estructura interna con naturalidad para lo cual ya no se **aplica para las estructuras de datos la ley de Demeter**.

Híbridos: Evitarlos, porque dificultan la inclusión de nuevas funciones y también de estructuras de datos. Son lo peor de ambos mundos.

Objetos de transferencia de Datos: La quintaesencia de una estructura de datos es una clase con variables públicas y sin funciones, estos son los famosos DTO (Data Transfer Object), útiles por su comunicación con la base de datos. Una forma especial de DTO son los **Registros activos pero que incluyen métodos de**

navegación como save y find. Desafortunadamente, muchos programadores intentan procesar estas estructuras de datos como si fueran objetos y les añaden métodos de reglas empresariales. **Evitar hacer esto, estaría creando híbridos** (usando: estructura de datos+objeto).

Conclusiones:

- Cuando en el sistema necesite añadir nuevos tipos de datos, usar objetos para esa parte del sistema.
- Cuando necesite añadir nuevos comportamientos, usar procedimientos (estructura de datos) en esa parte.

CAPITULO 7:

Procesar Errores: El control de errores es importante, pero si oscurece la lógica, es incorrecto. Siempre es recomendable generar una excepción al detectar un error, para que el código de invocación sea más limpio. Las excepciones comprobadas pueden ser útiles si tiene que crear una biblioteca crítica: tendrá que capturarlas. Pero en el desarrollo de aplicaciones generales, los costes de dependencia superan las ventajas.

Cuando se usen las excepciones proporcione el contexto adecuado para determinar el origen y la ubicación de un error.

Definir clases de excepción de acuerdo a las necesidades del invocador: Tratar de simplificar el código incluyendo la API adecuada en la invocación catch. Una buena practica es utilizar envoltorios que capturen y traduzcan excepciones generadas por la clase.

Definir el flujo normal: Recordar separación entre la lógica empresarial y control de errores.

Existen según Fowler **patrones de caso especial**, en donde se crean clases que procesen casos especiales. Para no tener que procesar excepciones en el código cliente. Dichos comportamientos se encapsulan en un objeto de caso especial.

Lo que se debe hacer en respuesta a la generación de `NULLPOINTEREXCEPTION` desde el interior de mi aplicación es quitar el exceso de comprobaciones de `null`. **Si siento la tentación de devolver null desde un método, pruebe generar una excepción o devuelva un objeto de caso especial.**

No pase NULL: No devuelva null desde los métodos, a menos que trabaje con una API que espere que pase null, debe evitarse siempre que sea posible.

Conclusiones:

- El código limpio es legible pero también robusto.
- El control de errores debe verse como algo independiente a nuestra lógica principal.

CAPITULO 8:

Límites: En este capítulo se ven prácticas y técnicas para definir con claridad los límites de nuestro software. Pasa con el ejemplo de `java.util.Map` que ofrece más prestaciones de las que necesitamos, por otro lado acepta todo tipo de datos y permite que cualquiera pueda borrarlo.

Explorar y aprender límites: Nuestra labor al utilizar código de terceros será hacerle pruebas. Aprender el código de terceros es complicado, y también integrarlo. Hacer ambas cosas al mismo tiempo es el doble de

complicado. Necesitamos crear pruebas que analicen nuestro entendimiento del código de terceros, pruebas que Jim Newkirk les llama prueba de aprendizaje. **Las pruebas se centrarán en lo queremos obtener de la API.** Sino hacemos las pruebas de aprendizaje que son las que demostraran que los paquetes se comportan como se espera, no hay garantías de que una vez integrados a nuestro código sea compatible con nuestras necesidades.

Usar código que todavía no existe: Cuando haga falta una parte del código (mundo desconocido) y queremos hacer todas las pruebas pertinentes al código que se desarrolla(mundo conocido), se recomienda crear una **Clase que simule** ese mundo desconocido.

Límites Limpios: Cuando usemos código que no controlamos, hay que prestar especial atención a proteger nuestra inversión y asegurarnos de que los cambios futuros no son demasiado costosos. **Debemos evitar que el código conozca los detalles de terceros.** Es más aconsejable depender de algo que controlemos que de algo que no controlemos, y menos todavía si nos controla. **Los límites de terceros se gestionan gracias a la presencia de puntos mínimos en el código que hagan referencia a los mismos. Podemos envolverlos como haríamos con Map** o usar un adaptador para convertir nuestra interfaz perfecta en la interfaz proporcionada.

CAPITULO 9:

Pruebas de Unidad: Desarrollo guiado por pruebas(DGP) es lo que se desarrolla en este capítulo. Los movimientos Agile y TDD han animado a muchos programadores a crear pruebas de unidad automatizadas y cada vez son más. Pero en esta alocada carrera por añadir pruebas a nuestra disciplina, muchos programadores han pasado por alto **dos de los aspectos más sutiles e importantes de diseñar pruebas de calidad.**

Las tres leyes del DGP: Esta ley establece que primero creemos las pruebas de unidad, antes del código de producción, pero tomando en cuenta las siguientes leyes:

- I. No crear código de producción hasta que haya creado una prueba de unidad que falle.
- II. No crear más de una prueba de unidad que baste como fallida y no compilar, se considera un fallo.
- III. No debe crear más código de producción que el necesario para superar la prueba de fallo actual.

El problema de dichas pruebas, es que al ser similar su tamaño con el de producción, puede suponer un problema de administración.

Realizar pruebas limpias: Cuando el código de prueba no se mantiene con los mismos estándares de calidad que su código de producción, **se convertirá en un gran problema** porque tener pruebas incorrectas es igual que no tenerlas. El problema es que las pruebas deben cambiar de acuerdo a la evolución del código y cuanto menos limpias sean, **más difícil es cambiarlas. Cuando más enrevesado sea el código de prueba, más probabilidades de que dedique más tiempo a añadir nuevas pruebas a la suite que el empleado en crear el nuevo código de producción.** Al modificar el código de producción, las pruebas antiguas comienzan a fallar y el desastre impide que las pruebas se superen, por lo que acaben por convertirse en un obstáculo interminable.

Por lo tanto, se puede concluir que el código de prueba es tan importante como el de producción. Requiere concentración, diseño y cuidado. Debe ser tan limpio como el código de producción.

Las pruebas propician posibilidades: Si las pruebas no son limpias, se perderán y sin ellas se pierde la flexibilidad del código, como también su mantenimiento y re-utilización.

Por tanto, tener una suite automatizada de pruebas de unidad que cubran el código de producción es la clave para mantener limpio el diseño y la arquitectura. **Las pruebas proporcionan las posibilidades, ya que permiten el cambio. Sin pruebas limpias perderemos las pruebas y el código se corromperá.**

¿Que hace que una prueba sea legible? Lo mismo que en el código: claridad, simplicidad y densidad de expresión. En una prueba se debe decir mucho con el menor número de expresiones posible.

El patrón **Generar-Operar-Comprobar** debe ser evidente en la estructura de las pruebas. La primera debe crear los datos de prueba, la segunda debe operar en dichos datos y la tercera comprobar que la operación devuelva los resultados esperados. **Esto logra eliminar detalles molestos. Las pruebas logran ser concisas y solo utilizan los tipos de datos y funciones que realmente necesitan.**

Lenguaje de pruebas específico del dominio: Las API de pruebas no se diseñan con antelación, sino que evolucionan con la refactorización continuada del código de prueba. Los programadores disciplinados siempre refactorizarán su código de prueba en versiones más breves y expresivas.

Un estándar Dual: Evitar el uso de *StringBuffer*, incluso en código de producción. Hay cosas que nunca se podrán hacer en un entorno de producción pero que sí se podrán en un entorno de prueba.

Una afirmación por Prueba: Existe una escuela de pensamiento que afirma que todas las funciones de prueba de prueba JUnit solo deben tener una instrucción de afirmación. Esto es, que las pruebas lleguen a una misma conclusión, que todos puedan entender de forma rápida y sencilla.

Los nombres de las funciones de la prueba deben ser hechas bajo la convención *dado-cuando-entonces*, para ser mejor leídas. Y si por casualidad, al dividir pruebas te encuentras con código duplicado se puede utilizar el patrón **Método de plantilla** e incluir las partes *dado/cuando* en la clase base, y las partes *entonces* en derivaciones diferentes o dejarlo así, porque al final son solo pruebas.

Puede haber más de una afirmación en un prueba, pero el número de afirmaciones debe ser mínimo.

Un solo concepto por Prueba: Es muy probable que la regla más indicada sea probar un único concepto en cada función de prueba, no extensas funciones que prueben una cosa diferente tras otra.

FIRST--> Las pruebas limpias siguen cinco reglas que son:

Fast = Las reglas deben ser rápidas y ejecutarse de forma rápida. Si ejecutan lentamente, ejecuta con menos frecuencia lo cual provoca la no detención de problemas con suficiente antelación para solucionarlos. **El código termina corrompiéndose al no poder limpiar el código.**

Independence = Una prueba no debe depender de otra, porque si falla una provocará efecto dominó y se dificulta el diagnóstico y ocultará efectos posteriores.

Repeat = Las pruebas deben poder repetirse en cualquier entorno. **Si no pueden ejecutarse en cualquier entorno, siempre habrá una excusa de su fallo.**

Self-Validating = Las pruebas deben tener un resultado booleano: o aciertan o fallan. **Si las pruebas no se validan automáticamente, el fallo puede ser subjetivo y la ejecución de las pruebas puede requerir una extensa evaluación manual.**

Puntuality: Deben crearse antes del código de producción que hace que acierten. Si se crean después, resultará difícil probarlos. **No diseñar código de producción que no se pueda probar.**

Conclusiones:

- Las pruebas son importantes para la salud del proyecto.
- Mejoran la flexibilidad, capacidad de mantenimiento y reutilización del código de producción.
- No permita que el código de pruebas se corrompan sino, sucederá lo mismo con el de producción.

CAPITULO 10:

La **organización** de las clases debe comenzar siempre con su lista de variables, así:

- Se declaran primero las constantes estáticas públicas si existen.
- Segundo las variables estáticas
- Tercero las variables de instancias privadas.
- Cuarto las funciones públicas
- Luego sus utilidades públicas invocadas tras la función que la invoca.

Encapsulación: Las variables y funciones de utilidad serán privadas **a menos que queramos que esa variable o función sea accesible para una prueba del mismo paquete en cuyo caso la definimos como *protected* o de paquete.**

Tamaño: Debe ser reducido, su medida irá de acuerdo a sus responsabilidades. Es decir, **el nombre de una clase deberá describir las responsabilidades que desempeña. EL NOMBRE SERÁ LA FORMA DE DETERMINAR EL TAMAÑO DE LA CLASE.** Cuanto más ambiguo sea el nombre de una clase, más probabilidades hay de que tenga demasiadas responsabilidades. **Debemos ser capaces de escribir una breve descripción de la clase en unas 25 palabras,** sin usar palabras como *sí, o y, pero.*

Esto es, básicamente el principio de SRP (Single Responsibility Principle) el cual indica que una clase o módulo debe tener uno y solo un motivo para cambiar. **Es decir, las clases deben tener una sola responsabilidad.** La identificación de responsabilidades (motivos del cambio) nos permiten reconocer y mejorar las abstracciones en nuestro código.

Reformulando lo anterior, se concluye que los sistemas deben estar formados por muchas clases reducidas, no por algunas grandes. Cada clase reducida incluirá(encapsulará) una única responsabilidad y ella colaborará con algunas otras para obtener el comportamiento deseado del sistema.

Cohesión: Las clases deberán tener un número reducido de variables de instancia. Por lo general cuantas más variables manipule un método, más cohesión tendrá con su clase. **El objetivo es que la cohesión de nuestras clases sea elevada.** Si lo logramos significa que los métodos y variables de la clase dependen unos de otros y actúan como un todo lógico. **Esto es consistencia.**

Organizar los Cambios: En muchos sistemas, el cambio es continuo y cada cambio es un riesgo que puede provocar que el sistema deje de funcionar de la forma esperada. **En un sistema limpio organizamos las clases para reducir los riesgos de los cambios.** Si la clase se considera totalmente lógica no debemos preocuparnos por separar las responsabilidades cuando esa clase sea demasiado grande, a menos que esa clase no necesite funcionalidad de actualización en el futuro.

Ejemplo de Refactorización:

Listado 10.9. Clase que debemos abrir para realizar cambios.

```
public class Sql {
    public Sql(String table, Column[] columns)
    public String create()
    public String insert(Object[] fields)
    public String selectAll()
    public String findByKey(String keyColumn, String keyValue)
    public String select(Column column, String pattern)
    public String select(Criteria criteria)
    public String preparedInsert()
    private String columnList(Column[] columns)
    private String valuesList(Object[] fields, final Column[] columns)
    private String selectWithCriteria(String criteria)
    private String placeholderList(Column[] columns)
}
```

Listado 10.10. Un grupo de clases cerradas.

```
abstract public class Sql {
    public Sql(String table, Column[] columns)
    abstract public String generate();
}

public class CreateSql extends Sql {
    public CreateSql(String table, Column[] columns)
    @Override public String generate()
}

public class SelectSql extends Sql {
    public SelectSql(String table, Column[] columns)
    @Override public String generate()
}

public class InsertSql extends Sql {
    public InsertSql(String table, Column[] columns, Object[] fields)
    @Override public String generate()
    private String valuesList(Object[] fields, final Column[] columns)
}

public class SelectWithCriteriaSql extends Sql {
    public SelectWithCriteriaSql(
        String table, Column[] columns, Criteria criteria)
    @Override public String generate()
}

public class SelectWithMatchSql extends Sql {
    public SelectWithMatchSql(
        String table, Column[] columns, Column column, String pattern)
    @Override public String generate()
}

public class FindByKeySql extends Sql {
    public FindByKeySql(
        String table, Column[] columns, String keyColumn, String keyValue)
    @Override public String generate()
}

public class PreparedInsertSql extends Sql {
    public PreparedInsertSql(String table, Column[] columns)
    @Override public String generate() {
    private String placeholderList(Column[] columns)
}

public class Where {
    public Where(String criteria)
    public String generate()
}

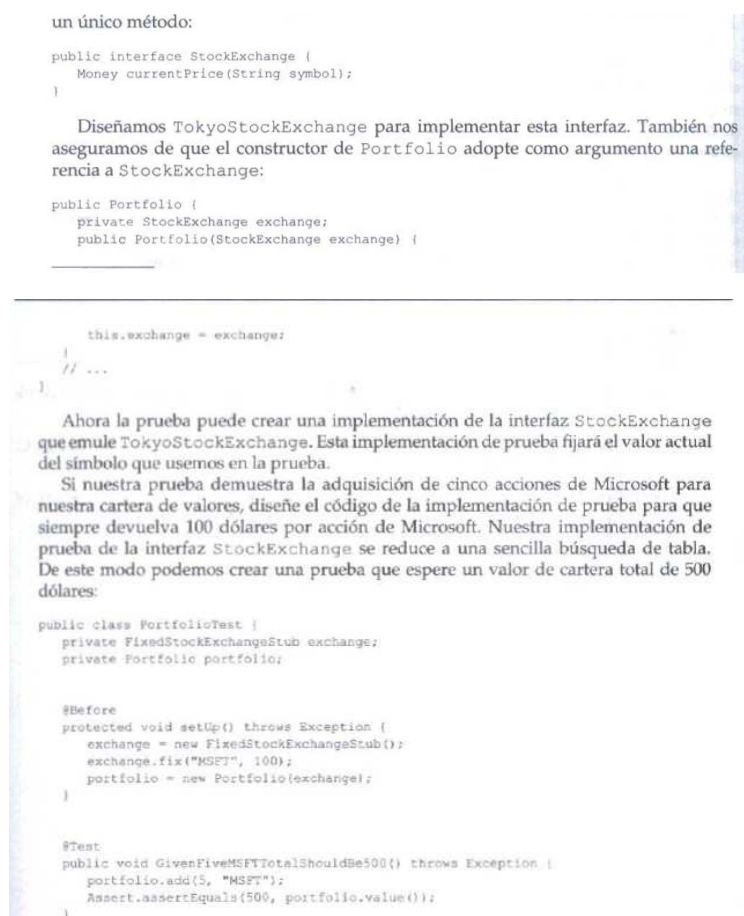
public class ColumnList {
    public ColumnList(Column[] columns)
    public String generate()
}
```

Esta refactorización logra que se cumpla el principio de SRP y también con otro principio clave del diseño de clases orientadas a objetos, denominado Principio abierto/cerrado: las clases deben abrirse para su ampliación para cerrarse para su modificación.

En un sistema ideal, incorporamos nuevas funciones ampliándolo, no modificando el código existente.

Aislarnos de los cambios: Las necesidades cambiarán y también lo hará el código. En la programación orientada a objetos aprendemos que hay clases concretas que contienen detalles de implementación (el código) y las clases abstractas que solo representan conceptos. Una clase cliente que dependa de detalles concretos está en peligro si dichos detalles cambian. Podemos recurrir a interfaces y clases abstractas para aislar el impacto de dichos detalles. **Las dependencias de detalles de concretos crean retos para nuestro sistema.**

Ejemplo de minimización de un sistema:



Al minimizar las conexiones de esta forma, nuestras clases cumplen otro principio de diseño: Principio de inversión de dependencias (DIP), el cual afirma que nuestras clases deben depender de abstracciones, no de detalles concretos.

CAPITULO 11:

Sistemas: Este capítulo muestra como mantener la limpieza en niveles superiores de abstracción, en el sistema.

Los sistemas de software deben separar el proceso de inicio, en el que se crean los objetos de la aplicación y

se conectan las dependencias, de la lógica de ejecución que toma el testigo tras el inicio.

El código de inicio no debe mezclarse con la lógica de tiempo de ejecución. Ejemplo:

```
public Service getService() {
    if (service == null)
        service = new MyServiceImpl(...); // ¿Lo bastante predeterminado
                                           // para la mayoría de los casos?
    return service;
}
```

En este caso no se podrá compilar sin antes resolver la dependencias que hay en MyServiceImpl, aunque nunca se llegue a usar este objeto en tiempo de ejecución. **Las pruebas también se ven afectadas porque se deberá asegurar de hacer es *test double* u objeto simulado al campo de servicio** antes de invocar este método en las pruebas de unidad por que MyServiceImpl es un objeto pesado.

El caso de inicialización tardía de los objetos como es el caso de MyServiceImpl es un problema, porque crea problemas de modularidad y duplicidad diseminados por todo el sistema. Y NO SE DEBE PERMITIR FALLOS DE MODULARIDAD, de lo contrario no tendremos sistemas robustos y bien formados.

Separar Main: Una forma de separar la construcción del uso consiste es trasladar todos los aspectos de la construcción a *main* o módulos invocados por main, y diseñar el resto del sistema suponiendo que todos los objetos se han creado y conectado correctamente. Existen otras soluciones como: Factorías Inyección de dependencias.

Evolucionar: Conseguir sistemas perfectos a la primera es un mito, dice Bob. Por el contrario, debemos implementar hoy, y refactorizar y ampliar mañana. Es la esencia de la agilidad iterativa e incremental. El desarrollo controlado por pruebas, la refactorización y el código limpio que generan hace que funcione a nivel del código.

Un bean de entidad es una representación en memoria de datos relacionales osea, una fila en una tabla.

Aspectos transversales: La arquitectura EJB2 se acerca a la verdadera separación de aspectos en determinados aspectos. Por ejemplo, comportamientos transaccionales, de seguridad entre otros se declaran en los descriptores de implementación, independientemente del código fuente. Por lo general intentará mantener todos sus objetos mediante la misma estrategia, con un determinado DBMS (oracle, Mysql , PostgreSQL...) y no archivos planos. **Para este aspecto las modificaciones de comportamiento no son invasivas para el código de destino.** Hay tres aspectos o mecanismos como este, tales como:

1. Proxies de Java: Son útiles en casos sencillos como envolver invocaciones de métodos en objetos o clases concretas. Sin embargo, el código es abundante y complejo por lo tanto inconveniente porque dificultan la ejecución de código limpio. Además, los proxies no ofrecen un mecanismo para especificar puntos de ejecución globales del sistema, imprescindibles para una verdadera ejecución AOP.
2. Estructuras AOP Java puras: Gran parte del código predefinido de proxy puede ser procesado de forma automática mediante herramientas como: Spring AOP y JBoss AOP. Se crea la lógica empresarial en forma de POJO, específicos de su dominio. **No dependen de estructuras empresariales (ni de otros dominios). Son más sencillos y fáciles de probar, por lo tanto garantiza que se implementen correctamente las historias, mantenimiento y evolución del código.**
3. Aspectos de AspectJ: **Esta herramienta-lenguaje es la más completa de separación a través de aspectos. Es una extensión de Java que ofrece compatibilidad de primer nivel para aspectos como construcciones de modularidad.** Los enfoques puros de Java proporcionados por Spring AOP y JBoss AOP son suficientes en el 80-90 por 100 de los casos en los que los aspectos son útiles. **Pero AspectJ ofrece un conjunto de herramientas avanzadas y compactas para la separación de aspectos.**

Aunque el software se rige por una física propia, es económicamente factible realizar cambios radicales si la estructura del software separa sus aspectos de forma eficaz. Esto significa que podemos iniciar un proyecto de software con una arquitectura simple pero bien desconectada, y ofrecer historias funcionales de forma rápida, para después aumentar la infraestructura. Evidentemente, no quiere decir que acometamos los proyectos sin timón. Debemos tener expectativas del ámbito general, objetivos y un programa, así como la estructura general del sistema resultante. Pero teniendo en cuenta que el sistema debe brindarme la posibilidad de cambiar de rumbo en el futuro.

Optimizar la Toma de decisiones: La modularidad y separación de aspectos permite la des-centralización de la administración y la toma de decisiones. Las decisiones deben ser pospuestas hasta el final, para poder tomarlas con la mayor cantidad de información posible.

La agilidad que proporciona un sistema POJO con aspectos modularizados nos permite adoptar decisiones óptimas a tiempo, basadas en los conocimientos más recientes. Además, se reduce la complejidad de estas decisiones.

Debemos recordar usar estándares sólo cuando añadan un valor demostrable, porque puede tardar demasiado crearlos y hay que tomar en cuenta las verdaderas necesidades de aquello para lo están dirigidas, **no hacerlo sólo por moda.**

Los sistemas necesitan lenguajes específicos del dominio: Un buen DSL minimiza el vacío de comunicación entre un concepto de dominio y el código que lo implementa, al igual que las prácticas ágiles optimizan entre un equipo y los accionistas del proyecto. Estos, aumentan el nivel de abstracción por encima del código y los patrones de diseño.

Conclusión: Si la agilidad se ve comprometida, la productividad sufre y las ventajas de TDD se pierden.

CAPITULO 12:

Limpieza a través de diseños Emergentes: Kent establece 4 reglas para lograr que un sistema sea un software bien diseñado:

I. Ejecuta todas las pruebas: El sistema debe actuar de forma prevista. **Un sistema puede tener un diseño perfecto sobre el papel pero si no existe una forma sencilla de comprobar que realmente funciona de la forma esperada, el esfuerzo sobre el papel es cuestionable.** Cuantas más pruebas creemos, más usaremos principios como DIP y herramientas con inyección de dependencias, interfaces y abstracción para minimizar dichas conexiones. **Nuestros diseños mejorarán más.**

II. No contiene duplicados: Refactorizamos para eliminar duplicados, un método bueno para lograrlo es el uso del patrón Método de plantilla.

III. Expresa la intención del programador: El objetivo es ver el nombre de una clase y función, y que sus responsabilidades no nos sorprendan. Otra forma es usar una nomenclatura estándar. Al usar los nombres de patrones estándar, como COMMAND o VISITOR, en los nombres de las clases que implementan dichos patrones pueden describir brevemente su diseño a otros programadores. **La forma más importante de ser expresivo es la práctica.**

IV. Minimiza el número de clases y métodos: Es la menos prioritaria de las cuatro, porque incluso conceptos básicos como la eliminación de código duplicado, la expresividad del código y SRP pueden exagerarse, en un esfuerzo por reducir el tamaño de las clases y métodos al mismo tiempo podemos estar creando demasiadas clases y métodos reducidos, por lo tanto **hay que tener cuidado con esto.**

CAPITULO 13:

Concurrente: Recordemos que la programación concurrente es la simultaneidad en la ejecución de múltiples tareas iterativas, **crear lo contrario (código que se ejecuta en el mismo proceso) es mucho más sencillo. Pero aunque en principio funcione correctamente, comenzará a dar problemas cuando el sistema se someta a determinadas presiones. AQUÍ ES DONDE NACE LA NECESIDAD DE LA PROGRAMACIÓN CONCURRENTE.**

La concurrencia es una estrategia de desvinculación. En aplicaciones de un solo proceso, el qué y el cuándo están tan firmemente vinculados que el estado de la aplicación se puede determinar analizando la huella de la pila. **La desvinculación del qué del dónde puede mejorar considerablemente el rendimiento y la estructura de una aplicación. Desde un punto de vista estructural, la aplicación parece una serie de equipos colaboradores y no un gran bucle principal.**

Para poder desarrollar sistemas concurrentes libres de problemas debemos aplicar los siguientes principios:

1. SRP: Establece que un método, clase o componente debe tener una sola responsabilidad. Por lo tanto el diseño de concurrencia deberá separarse del código de producción(o resto del código).
2. Limite el ámbito de los datos: Porque dos procesos que modifiquen el mismo campo u objeto compartido pueden interferir entre ellos y provocar comportamientos inesperados(incorrectos). Solución: **Use la palabra synchronized** para proteger una sección importante del código que use el objeto compartido, aunque conviene limitar la cantidad de estas secciones, Es decir encapsule los datos y limite el acceso a los datos compartidos.
3. Use copias de datos: Si existe una forma sencilla de evitar los objetos compartidos, el código resultante tendrá menos problemas. Puede que le preocupe el coste de la creación de objetos adicionales, pero merece la pena experimentar si es un problema real. Si todo esto evita la sincronización, las ventajas de evitar el bloque quedarán compensadas con las creaciones adicionales y la sobrecarga de la recolección de elementos sin usar.
4. Que los procesos sean independientes: Cada proceso deberá procesar una solicitud cliente y todos los daos necesarios provendrán de un origen sin compartir y se almacenarán como variables. Así los procesos se comportarán como si fueran los únicos en el mundo y no existiera requisitos de sincronización.
5. Conozca las bibliotecas: Revise las clases que tenga: `java.util.concurrent`, `java.util.concurrent.atomic` y `java.util.concurrent.locks`
6. Conozca los modelos de ejecución: Recursos vinculados, Exclusión mutua, Inanición, bloqueo, bloqueo activo.
7. Probar código con Procesos: Demostrar que el código es correcto no resulta práctico. Las pruebas no garantizan su corrección. Sin embargo, las pruebas adecuadas pueden minimizar los riesgos en aplicaciones de un solo proceso. Lo que se recomienda es crear pruebas que puedan detectar problemas y ejecutarlas periódicamente, en distintas configuraciones de programación y del sistema, y cargas.
8. Considerar los fallos como posibles problemas de los procesos: No como algo aislado.
9. El código con procesos se debe poder conectar a otros elementos y debe ser modificable.
10. Ejecutar en diferentes plataformas: Porque cada sistema operativo tiene una política de procesos diferente que afecta a la ejecución del código. El código con procesos múltiples se comporta de forma distinta en cada entorno.
11. Diseñar el código para probar y forzar fallos: Las pruebas sencillas no suelen mostrar los fallos del

código concurrente, para aumentar las posibilidades de capturarlos se recomienda usar métodos como: `Object.wait()`, `Object.sleep()`, `Object.yield()` y `Object.priority()`.

Conclusiones:

- Es complicado conseguir código correcto.
- Aprenda a localizar regiones del código que se puedan bloquear y bloqueélas. Evitar llamar esa zona desde otra.
- Los problemas se acumularán. Los que no aparezcan inicialmente suelen considerarse esporádicos y suelen producirse en la fase de carga o de modo aparentemente aleatorio.
- La capacidad de prueba, si aplica las tres leyes de TDD le ofrecerá la compatibilidad necesaria para ejecutar código en distintas configuraciones.

CAPITULO 14:

Refinamiento Sucesivo: Java al ser un lenguaje de tipos estáticos, requiere muchas palabras para satisfacer el sistema de tipos a la hora de hacer un programa, a diferencia de otros como Python.

Cuando se encuentre que ha creado un programa con muchos tipos diferentes y todos con métodos similares, refactorice. Para evitar que su programa deje de funcionar después de refactorizarlo-mejorarlo recurra a la disciplina TDD(Test-Driven Development) . **Una de las doctrinas centrales de este enfoque es mantener la ejecución del sistema en todo momento. Es decir, que he de crear pruebas automatizadas que ejecuten rápidamente, verificando que el comportamiento del sistema es el mismo, después de los cambios.**

Las pruebas de unidad se suelen crear en Java y se administran con JUnit. Las pruebas de aceptación se crean como páginas de wiki en FitNesse.

Con el incrementalismo, se deben realizar pruebas individualmente, para saber cuando hay fallos cuál prueba falla y asegurarse que esa prueba es correcta antes de seguir con el siguiente cambio.

Se debe separar el código de excepciones y de error del módulo donde tenemos el código original.

El diseño de software correcto se basa en gran parte en las particiones, en crear zonas adecuadas para incluir distintos tipos de código. Esta separación hace que el código sea más fácil de entender y mantener.

Conclusiones:

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. No hay nada que afecte más negativamente a un proyecto de desarrollo que el código incorrecto. Es verdad, que el código incorrecto puede limpiarse pero será muy costoso sino se hace antes tanto refactorización como pruebas. La falta de tiempo no debe ser una excusa para no corregir el código, si no se cumple con las condiciones anteriormente planteadas su código tiene como destino el desastre.

CAPITULO 15:

Aspectos internos de JUnit: JUnit es una de las estructuras de Java más conocidas. De concepción sencilla, definición precisa y documentación elegante.

Se estudia en este capítulo el fragmento de código llamado `ComparisonCompactor`, que permite identificar errores de comparación de cadenas, creado por dos genios; Kent Beck y Eric Gamma.

El código es refactorizado, para obedecer la Regla del código limpio quitando redundancias y usando nombres exclusivos, reduciendo el tamaño de los métodos y creando nuevos, mejor renombrados. Todo es re-ordenado topo-lógicamente. Se crea un grupo de funciones análisis y otro de funciones síntesis dentro del módulo creado por el par de autores mencionados anteriormente y se elimina un if que estaba de más.

CAPITULO 16:

Refactorización de SerialDate: Se analiza la clase **SerialDate** que se encuentra dentro del paquete **org.jfree.date** de la biblioteca **JCommon** del autor David Gilbert.

Esta clase representa una fecha en Java, para solucionar el problema de tener fechas en lugar de tener fechas con limitaciones de zonas horarias.

Bob corrige y añade multitud de casos de prueba ya que los iniciales originales no se superaban, se eliminan comentarios y se ponen todos esos comentarios en un sólo lenguaje Javadoc, en lugar de usar HTML, Inglés y Javadoc. Cambia el nombre de la clase *SerialDate* por *DayDate*, porque lo considera más apropiado al ser una clase abstracta que trabaja con días.

Encuentra dilemas como que siendo una clase abstracta muestre información de implementación, contaminando la clase. Porque según Bob ese tipo de información (la de implementación) se debe obtener mediante una variante de instancia.

Para corregirlo, propone usar el patrón de factoría abstracta y crear *DayDateFactory*, dicha factoría crea las instancias de *DayDate* que se necesitan y también responde a preguntas sobre la implementación, como las fechas máxima y mínima.

La clase resultante factoría sustituye los métodos *createInstance* por métodos *makeDate*, mejorando los nombres. Los métodos estáticos delegados en métodos abstractos usan una **combinación de los patrones de instancia única, decorator y factoría abstracta**.

Elimina las palabras clave *final* de argumentos y declaraciones de variables, ya que no parecen servir de mucho. La eliminación de *final* por la totalidad del código. Según Bob, no está de acuerdo con diseminar *final* por todo el código porque existen casos donde se puede usar como constante ocasional, pero que en general, esta palabra clave apenas añade valor y suele ser un estorbo. **También elimina instrucciones if y for duplicadas porque las ve irrelevantes. Simplifica los algoritmos complicados. Cambió varios métodos estáticos a métodos de instancia.**

Al limpiar el código, el alcance de las pruebas aumenta. Bob corrigió según él, algunos errores y aclaró y redujo el código.

CAPITULO 17:

Síntomas y Heurística: Bob sigue dando algunos consejos útiles como el que los comentarios deben reservarse para notas técnicas sobre el código y el diseño, **no deben estar diseminados por todo el código, a esto se le llama Información Inapropiada.**

También deben eliminarse los comentarios obsoletos. Debemos eliminar comentarios redundantes como la firma de una función. Ejemplo:

```
i++; // incrementar i
```

Otro ejemplo es un Javadoc que no dice más (o incluso menos) que la firma de una función:

```
/**
 * @param sellRequest
 * @return
 * @throws ManagedComponentException
 */
public SellResponse beginSellItem(SellRequest sellRequest)
    throws ManagedComponentException
```

Los comentarios deben comunicar lo que el código no pueda expresar por sí mismo.

Si es necesario escribir un comentario, debemos ser breves y usar buena gramática. Si tenemos código comentado, habrá que eliminarlo también.

La generación requiere más de un paso: La generación de un proyecto deberá ser una operación sencilla. Sin comprobaciones de demasiados elementos de control del código fuente.

Las pruebas requieren más de un paso: Las pruebas de unidad deben ser ejecutadas con un sólo comando. Deberá ser algo rápido, sencillo y obvio.

Los Argumentos de salida: estos argumentos son ilógicos. El lector espera que los argumentos sean de entrada no de salida. Si se debe cambiar el estado de algo, el objeto debe ser cambiado en la función que se invoca.

Argumentos booleanos: El tenerlos significa que la función hace más de una cosa. Deben ser eliminados.

Funciones muertas: Los métodos que nunca son invocados deben eliminarse.

Varios lenguajes en un archivo: Debemos reducir al mínimo el tener varios lenguajes de programación en el mismo código.

Duplicación: Esta regla la llama DRY (Don't repeat Yourself). **Siempre que se vea en el código alguna duplicación significa que esa duplicación podría convertirse en una subrutina o en otra clase. Al hacer esto, aumentamos el nivel de abstracción volviendo el código más rápido y menos proclive a errores. CREE MÉTODOS SIMPLES.**

Cuando tengamos cadenas *switch/case* o *if/else* repetidos en diversos módulos, **probando las mismas condiciones, REEMPLACELAS POR POLIMORFISMO.**

Los niveles de Abstracción del código: Todos los conceptos de nivel superior deben estar en la clase base, y los conceptos de nivel inferior deben estar en las variantes. Es decir, constantes, variables o funciones de utilidad que solamente pertenezcan a la implementación detallada no deben aparecer en la clase base. **La clase base no debe saber nada al respecto de estos elementos. Esta regla también se aplica a archivos fuente, componentes y módulos.**

El aislamiento de abstracciones es una de las operaciones más complicadas para los desarrolladores de software y no se puede corregir cuando se realiza de forma incorrecta.

Clases bases que dependen de sus variantes: Por lo general, las clases base no deben saber nada de sus derivadas.

Exceso de Información: Los módulos bien definidos tienen interfaces reducidas que nos permiten hacer mucho con poco. **Una interface bien definida no ofrece demasiadas funciones y las conexiones son reducidas.**

Recordar que:

- a) **Cuanto menos métodos tenga una clase, mejor.**
- b) **Cuanto menos variables conozca una función, mejor.**

- c) **Cuanto menos variable de instancia tenga una clase, mejor.**
- d) **Oculte sus datos**
- e) **Oculte sus funciones de utilidad**
- f) **Oculte sus constantes y elementos temporales**
- g) **No cree multitud de variables y funciones protegidas para sus subclases**
- h) **Limite la información para reducir las conexiones.**

Separación vertical: Variables y funciones deben ser definidas cerca de donde se utilicen. Las funciones privadas deben definirse justo debajo de su primer uso. Para localizar una función privada debe bastar con buscar debajo de su primer uso.

Conexiones artificiales: Los elementos que no dependen unos de otros no deben conectarse de forma artificial. Por ejemplo no se deben declarar funciones static de propósito general en clases específicas. **Piense en dónde deben declararse sus funciones, constantes y variables. No las deje en el punto más cómodo.**

Argumentos de Selector: No deben ser boolean, los selectores que determinen el comportamiento de una función, mejor usar otro tipo de argumentos, por ejemplo enteros.

Intención Desconocida: Que tu código sea lo más expresivo posible, nada de notación Húngara y números mágicos por aquí y por allá.

Elementos estáticos: Por lo general es mejor decantarse por métodos no estáticos. En caso de tener dudas, convierta la función en no estática. Si quiere que su función sea estática asegúrese que nunca querrá que sea polimórfica.

Comprender el Algoritmo: Antes de creer que hemos terminado con una función, asegúrese de que entiende su funcionamiento. Para saber esto, lo mejor es refactorizar el código.

Usar polimorfismo antes que If/Else o Switch/Case: Los casos de instrucciones switch deben crear objetos Polimórfico que ocupen el lugar de otras instrucciones switch similares en el resto de sistema.

Sustituya los números mágicos por constantes dentro del código

Precisión: No debemos ser indolentes. Si invocamos una función que puede devolver null, asegurarnos de comprobar ese null. Si existe la posibilidad de una actualización concurrente, asegurarnos que implementamos algún tipo de mecanismo de bloqueo. No cree código ambiguo.

Evite las condiciones negativas: Es más fácil de entender su código si usa las positivas.

Separación de Niveles de Abstracción: Una de las tareas más importantes de la refactorización, y también una de las más complejas. Las instrucciones de una función deben crearse en el mismo nivel de abstracción, un nivel por debajo de la operación descrita por el nombre de la función.

Mantener los datos configurables en los niveles superiores.

Cree código Silencioso: Esto es la ley de Demeter, donde tengo tres módulos A, B, C; el módulo A, no tiene porque saber lo que hace C, si A solo colabora con B, y B con C.

Importe paquetes comodín: No largas listas de 80 líneas de importaciones.

No herede constantes: Es horrible. No oculte constantes haciendo uso de herencia para burlar las reglas de ámbito del lenguaje. Usemos importaciones estáticas.

Constantes frente a enumeraciones: Si puede usar enum en lugar de int, úselos.

Elegir nombres en el nivel correcto de abstracción: No elija nombres que comuniquen implementación; seleccione nombres que reflejen el nivel de abstracción de la clase o la función con la que trabaje. De nuevo, siempre que realice una pasada por su código, es probable que encuentre una variable con nombre en un nivel demasiado bajo. Cambie esos nombres cuando los vea. **Para que el código sea legible necesita mejora continua.**

Usar nombres extensos para ámbitos extensos: La longitud de un nombre debe estar relacionado con la de su ámbito.

Usar una herramienta de cobertura: Las herramientas de cobertura indican vacíos en su estrategia de pruebas. Facilitan la detección de módulos, clases y funciones insuficientemente probadas.

Una prueba ignorada es una pregunta sobre ambigüedad: Podemos expresar una duda del detalle de un comportamiento o sobre los requisitos en forma de una prueba comentada o como prueba anotada con *@ignore*.

Se debe probar las condiciones límite.

Probar de forma exhaustiva los errores: Porque cuando se encuentra uno dentro de una función, seguramente no será el único error.

Las pruebas deben ser rápidas: Una prueba lenta no se ejecuta. Cuando se ponen feas, las pruebas lentas se eliminan de la suite. Por lo tanto, intentaremos que las pruebas sean rápidas.