Checklist prácticas de construcción de software

Nota previa a la revisión:

Es importante dejar en claro que aquellas practicas que se encuentre en amarillo, son aquellas practicas que no se realizaron porque, quizá entraban en conflicto con alguna otra o no eran necesarias, ya sea porque no utilizábamos cosas que se debían refactorizar, o por el hecho de que no consideramos que nuestro proyecto necesitara, por ejemplo:

No aplicamos la convención de java con respecto a las sentencias for, debido a que no utilizamos esta sentencia durante la codificación.

Aclarado esto, estas son las practicas aplicadas a nuestro proyecto.

Convenciones de código Java

Comentarios de comienzo	Todos los ficheros deben comenzar con un comentario en el que se lista el nombre de la clase, información de versión, fecha y copyright
Sentencias package e import	La primera línea no-comentario de los ficheros fuente Java es la sentencia package. Después de esta, pueden seguir varias sentencias import.
Declaraciones de clases e interfaces	Este es el orden en el que deben aparecer las partes de la declaración de una clase o interfaz: 1. Comentario de la documentación de la clase o interfaz 2. Sentencia class o interface 3. Comentario de implementación de la clase o interfaz si fuera necesario 4. Variables de clase (static) 5. Variables de instancia 6. Constructores 7. Métodos
Longitud de la línea	Evitar las líneas de más de 80 caracteres.
Rompiendo líneas	Cuando una expresión no entre en una línea, romperla de acuerdo a los siguientes principios: • Romper después de una coma • Romper antes de un operador • Preferir roturas de alto nivel que de bajo nivel

	• Alinear la nueva línea con el comienzo de la expresión al mismo nivel de la línea anterior • Si las reglas anteriores llevan a un código confuso o a código que se aglutina en el margen derecho, indentar justo 8 espacios en su lugar
Formatos de los comentarios de implementación	Los programas pueden tener cuatro estilos de implementación: 1. **Comentarios de bloque:** Se usan para las descripciones de ficheros, métodos, estructuras de datos y algoritmos, estos se podrán usar al comienzo de cada fichero o antes de cada método, al igual que en el interior de los métodos. Este tipo de comentarios en el interior de una función deben ser indentados al mismo nivel que el código que describen. 2. **Comentarios de una línea:** Pueden aparecer comentarios cortos de una única línea al nivel de código que siguen, si un comentario no se puede escribir en una línea, debe seguir el formato de comentarios de bloque. Un comentario de una sola línea debe ir precedido de una línea en blanco. 3. **Comentarios de remolque:** Pueden aparecer comentarios muy pequeños en la misma línea que describen, pero deben ser movidos lo suficientemente lejos para separarlos de las sentencias. Si mas de un comentario corto aparecen en el mismo trozo de código, deben ser indentados con la misma profundidad. 4. **Comentarios de fin de línea:** El delimitador de comentario “//” puede convertir en comentario una línea completa o una parte de una línea. No debe ser usado para

	hacer comentarios de varias líneas consecutivas; sin embargo, puede usarse en líneas consecutivas para comentar secciones de código.
Comentarios de documentación	Estos comentarios describen clases Java, interfaces, constructores, métodos y atributos. Cada comentario de documentación se encierra con los delimitadores de comentarios “/**…*/”, con un comentario por clase, interface o miembro (método o atributo). Si se necesita dar información sobre una clase, interface, variable o método que no es apropiada para la documentación, usar un comentario de implementación de bloque. Este tipo de comentarios no deben colocarse en el interior de la definición de un método o constructor, ya que Java asocia los comentarios de documentación con la primera declaración después del comentario.
Cantidad de declaraciones por línea	Se recomienda una declaración por línea, ya que facilita los comentarios. No poner diferentes tipos en la misma línea.
Inicialización de variables	Intentar inicializar las variables locales donde se declaran. La única razón para no inicializar una variable donde se declara es si el valor inicial depende de algunos cálculos que deben ocurrir.
Colocación de declaraciones	Poner las declaraciones solo al principio de los bloques. No esperar al primer uso para declararlas; puede confundir a programadores no preavisados y limitar la portabilidad del código dentro de su ámbito de visibilidad. La excepción a la regla son los índices de bucles “for”.
Declaraciones de class e interfaces	Al codificar clases e interfaces de Java, se siguen las siguientes reglas de formato: • Ningún espacio en blanco entre el nombre de un método y el paréntesis que abre su lista de parámetros

	• La llave de apertura aparece al final de la misma línea de la sentencia declaración • La llave de cierre empieza una nueva línea indentada para ajustarse a su sentencia de apertura correspondiente, excepto cuando no existen sentencias entre ambas, que debe aparecer inmediatamente después de la apertura • Los métodos se separan con una línea en blanco
Sentencias simples	Cada línea debe contener como mucho una sentencia
Sentencias compuestas	Son sentencias que contienen listas de sentencias encerradas entre llaves: • Las sentencias encerradas deben indentarse a un nivel mas que la sentencia compuesta • La llave de apertura se debe poner al final de la línea que comienza la sentencia compuesta; la llave de cierre debe empezar una nueva línea y ser indentada al mismo nivel que el principio de sentencia compuesta • Las llaves se usan en todas las sentencias, incluso las simples, cuando forman parte de una estructura de control, como en las sentencias if-else o for. Esto hace mas sencillo añadir sentencias sin incluir bugs accidentalmente por olvidar las llaves.
Sentencias de retorno	Una sentencia return con un valor no debe usar paréntesis a menos que hagan el valor de retorno mas obvio de alguna manera.
Sentencias if, if-else, if else-if else	Siempre deben llevar las llaves de inicio y final, y siguiendo las convenciones de declaración de sentencias.
Sentencias for	Una sentencia for debe estar compuesta de 3 sentencias y en el siguiente orden: incialización, condición, actualizacion, seguido de sus llaves de inicio y cierre

	(en caso de tener sentencias dentro) o punto y coma (en caso de que sea vacia). Al usar el operador coma en la clausula de inicialización o actualización de una sentencia for, evitar la complejidad de usar mas de tres variables. Si se necesita, usar sentencias separadas antes del bucle for o al final del bucle.
Sentencias while	Una sentencia while debe incluir como sentencia la condición y debe estar seguida por sus respectivas llaves de inicio y final o punto y coma en caso de ser vacía.
Sentencias do-while	Una sentencia do-while debe iniciar con la palabra “do” seguida de sus llaves de apertura y cierre y dentro de estas las sentencias correspondientes, después de la llave de cierre debe estar la sentencia while con la condición dentro.

Sentencias switch	Una sentencia switch debe tener la siguiente forma:
	switch (condicion) {
	case ABC:
	sentencias;
	/* este caso se propaga */
	case DEF:
	sentencias;
	break;
	case XYZ:
	sentencias;
	break;
	default:
	sentencias;
	break;
	}
	Cada vez que un caso se propaga (no incluye la sentencia break), añadir un comentario donde la sentencia break se encontraría normalmente, en el ejemplo anterior el caso se propaga.
	Cada sentencia switch debe incluir un caso por defecto. El break en el caso por defecto es redundante, pero prevee que se propague por error si luego se añade otro caso

Sentencias try-catch	Una sentencia try-catch debe tener la siguiente forma:

	try {
	sentencias;
	} catch (Exception class e) {
	Sentencias;
	}
	Una secuencia try-catch puede ir seguida de un finally, cuya ejecución se ejecutará independientemente de que el bloque try se haya completado con éxito o no.
	try {
	sentencias;
	} catch (Exception class e) {
	Sentencias;
	} finally {
	Sentencias;
	}

Lineas en blanco	Las líneas en blanco mejoran la facilidad de lectura separando secciones de código que están lógicamente relacionadas. Se deben usar siempre en dos líneas en blanco en las siguientes circunstancias: • Entre las secciones de un fichero fuente • Entre las definiciones de clases e interfaces Se debe usar siempre una línea en blanco en las siguientes circunstancias: • Entre métodos. • Entre las variables locales de un método y su primera sentencia. • Antes de un comentario de bloque o de un comentario de una línea • Entre las distintas secciones lógicas de un método para facilitar la lectura.
Espacios en blanco	Se deben usa espacios en blanco en las siguientes circunstancias: • Una palabra clave del lenguaje seguida por un paréntesis debe separarse por un espacio • Debe aparecer un espacio en blanco después de cada coma en las listas de argumentos.

	• Todos los operadores binarios excepto "." se deben separar de sus operadores con espacios en blanco. Los espacios en blanco no deben separar los operadores unarios, incremento y decremento de sus operandos. • Las expresiones en una sentencia for se deben separar con espacios en blanco. • Los "Casts" deben ir seguidos de un espacio en blanco.
Convenciones de nombres	Tipos de identificadores: • **Paquetes:** El prefijo del nombre de un paquete se escribe siempre con letras ASCII en minúsculas, y debe ser uno de los nombres de dominio de alto nivel, actualmente com, edu, gov, mil, net, org, o uno de los codigos ingleses de dos letras que identifican cada país como se especifica en el ISO Standard 3166, 1981. Los subsecuentes componentes del nombre del paquete variarán de acuerdo a las convenciones de nombres internas de cada organización. • **Clases:** Los nombres de las clases deben ser sustantivos, cuando son compuestos tendrán la primera letra de cada palabra que lo forma en mayúsculas. Intentar mantener los nombres de las clases simples y descriptivos. Usar palabras completas, así evitar acrónimos y abreviaturas (a no ser que la abreviatura sea mucho mas conocida que el nombre completo, como URL o HTML). • **Interfaces:** Los nombres de las interfaces siguen la misma regla que las clases. • **Métodos:** Los métodos deben ser vistos como verbos, cuando son compuestos tendrán la primera

	letra en minúscula, y la primera letra de las siguientes palabras que lo forma en mayúscula. • **Variables:** Excepto las constantes, todas las instancias y variables de clase o método empezarán con minúscula. Las palabras internas que lo forman (si son compuestas) empiezan con su primera letra en mayúsculas. Los nombres de variables no deben empezar con los caracteres subguión "_" o signo de dólar "$", aunque ambos están permitidos por el lenguaje. Los nombres de las variables deben ser cortos, pero con significado. La elección del nombre de una variable debe ser mnemónico, designado para indicar a un observador casual su función. • **Constantes:** Los nombres de las variables declaradas como constantes deben ir totalmente en mayúsculas separando las palabras con subguión ("_").
Proporcionando acceso a variables de instancia y de clase	No hacer ninguna variable de instancia o clase publica sin una buena razón. A menudo las variables de instancia no necesitan ser asignadas/consultadas explícitamente, a menudo esto sucede como efecto lateral de llamadas a métodos.
Referencias a variables y métodos de clase	Evitar usar un objeto para acceder a una variable o método de clase (static). Usar el nombre de la clase en su lugar.
Constantes	Las constantes numéricas (literales) no se deben codificar directamente, excepto -1, 0 y 1, que pueden aparecer en un bucle for como contadores.
Asignaciones de variables	• Evitar asignar el mismo valor a varias variables en la misma sentencia. Es difícil de leer. • No usar el operador de asignación en un lugar donde se pueda confundir con el de igualdad.

	• No usar asignación embebida como un intento de mejorar el rendimiento en tiempo de ejecución, ese es trabajo del compilador.
Paréntesis	En general, es una buena idea usar paréntesis en expresiones que implican distintos operadores para evitar problemas con el orden de precedencia de los operadores. Incluso si parece claro el orden de precedencia de los operadores, podría no ser así para otros, no se debe asumir que otros programadores conozcan el orden de precedencia.
Valores de retorno	Intentar hacer que la estructura del programa se ajuste a la intención.
Expresiones antes de '?' en el operador condicional	Si una expresión contiene un operador binario antes de '?' en el operador ternario '?:', se debe colocar entre paréntesis.
Comentarios especiales	Usar XXX en un comentario para indicar que algo tiene algún error pero funciona. Usar FIXME para indicar que algo tiene algún error y no funciona.

Checklist de buenas prácticas referente a variables en construcción de Software

Deshabilitar declaraciones implícitas
Declarar todas las variables
Utilizar convenciones de nombramiento
Revisar los nombres de las variables
Inicializar cada variable cuando sea declarada
Inicializar cada variable cerca de donde sea usada por primera vez
De preferencia, declarar y definir cada variable cerca de donde sea usada
Poner especial atención a contadores y acumuladores
Inicializar los miembros de datos de las clases mediante constructores
Revisar la necesidad de reinicialización de las variables
Inicializa las constantes nominales una vez, inicializa las variables con el código ejecutable
De ser posible, utilizar las características de inicialización automática del compilador
Tomar ventaja de los mensajes de advertencia del compilador
Revisar que los valores de los parámetros de entrada sean válidos
Usar verificadores de acceso de memoria para identificar apuntadores malos
Identificar las referencias a las variables

Mantener lo menos posible el tiempo de vida de las variables
Inicializar las variables utilizadas en un bucle inmediatamente antes del bucle y no al comienzo de la rutina que contiene al bucle
No asignar un valor a una variable sino justo antes de que se use el valor
Agrupar sentencias relacionadas
Comenzar con la visibilidad más restringida y ampliar el alcance de la variable solo si es necesario
Revisar la persistencia de las variables
Ligar las variables en los momentos adecuados, logrando un balance entre la flexibilidad de un enlace tardío y el aumento de la complejidad asociada a este enlace tardío. Estos son algunos ejemplos de los tiempos en que una variable puede vincularse a un valor: • Tiempo de codificación (uso de números mágicos). • Tiempo de compilación (uso de una constante nominal). • Tiempo de carga (lectura de un valor de una fuente externa como el registro de Windows) • Tiempo de instanciación de objeto (como leer el valor cada vez que es creada). • Justo a tiempo (como leer el valor cada vez que se dibuja la ventana)
Corresponder los tipos de datos a las estructuras de control: • Los datos secuenciales se traducen en declaraciones secuenciales en un programa • Los datos selectivos se traducen en declaraciones if y case en un programa • Los datos iterativos se traducen en estructuras for, repeat y while en un programa.
Utilizar cada variable para un único propósito
Evitar las variables con significados ocultos (variables que tienen otros significados)
Asegurar que todas las variables declaradas son usadas

Checklist de refactorings

Extract Method	• **Razón de refactorización:** Se tienen fragmentos de código que pueden ser agrupados en métodos • **Señales de aplicación:** Un método es demasiado largo. Un código necesita comentarios para poder entender su comportamiento. • **Solución:** Crear un nuevo método y nombrarlo por lo que hace, no por como lo hace
Inline Method	• **Razón de refactorización:** El cuerpo de un método muy

	pequeño que es tan claro como su nombre. • **Señales de aplicación:** El cuerpo del método es tan claro como su nombre. Se tiene un grupo de métodos que parecen mal factorizados. Puede incorporarlos a todos en un gran método (Inline Method) y luego volver a extraer los métodos (Extract Method) • **Solución:** Incluir en el cuerpo del método dentro del cuerpo del método que lo llama y posteriormente eliminar el método llamado.
Inline Temp	• **Razón de refactorización:** Se tiene una variable temporal a la que se le asigna el valor de una llamada a método. Con frecuencia esta temporal no tiene otra funcionalidad, por lo que es fácil eliminarla. • **Señales de aplicación:** A una variable temporal se le asigna el valor de una llamada a método y se interpone en el camino de otras refactorizaciones. • **Solución:** Eliminar la variable temporal.
Replace Temp with Query	• **Razón de refactorización:** Se está utilizando una variable temporal para almacenar el resultado de una expresión. El problema con las variables temporales radica en que solo tienen visibilidad temporal y local, sin embargo, si se tuvieran varios métodos requiriendo esta información, sería conveniente reemplazar esa variable temporal por un método que pudiera ser llamado cuando sea necesario. • **Solución:** Se debe extraer la expresión en el método. Reemplazar todas las referencias

	de la variable temporal con un método. De esta manera el nuevo método podrá ser usado en otros métodos
Introduce Explaining Variable	• **Razón de refactorización:** Se tiene una expresión complicada. Algunas expresiones pueden ser muy complejas o difíciles de leer. En estas situaciones las variables temporales pueden ayudar a romper la expresión en partes más entendibles. • **Señales de aplicación:** A una variable temporal se le asigna el valor de una llamada a método y se interpone en el camino de otras refactorizaciones. • **Solución:** Se puede poner el resultado de la expresión, o partes de la expresión, en una variable temporal con un nombre que explique su propóstio.
Split Temporary Variable	• **Razón de refactorización:** Se tiene una variable temporal asignada más de una vez, pero no es una variable de bucle, ni tampoco una variable temporal recolectora. • **Señales de aplicación:** Cualquier variable temporal con más de una responsabilidad debe ser reemplazada por una variable temporal para cada responsabilidad. Usar una variable temporal para dos cosas diferentes resulta muy confuso para el lector. • **Solución:** Para resolver la situación se debe hacer una variable temporal diferente para cada asignación.
Remove Assignments to Parameters	• **Razón de refactorización:** Se asigno código a parámetros. • **Señales de aplicación:** En Java el pase de parámetros es por valor, cualquier cambio en el

	parámetro no se refleja en el resultado fuera de la rutina. Esto puede ser confuso. Por otro lado, el código es más claro si se usa el parámetro para representar el dato que ha sido pasado al método. Ya que es consistente con su uso. Estas reglas no necesariamente son las mismas en otros lugares • **Solución:** Se deben utilizar variables temporales para eliminar la asignación de código de parámetros.
Replace Method with Method Object	• **Razón de refactorización:** Se tiene un método extenso que utiliza variables locales de tal manera que no se puede aplicar la práctica de refactorización Extract Method. • **Solución:** Este detalle se resuelve convirtiendo el método en su propio objeto, de manera que todas las variables locales se conviertan en atributos de ese objeto. Posteriormente, se puede descomponer el método a otros métodos en el objeto creado.
Substitute Algorithm	• **Razón de refactorización:** Se requiere reemplazar un algoritmo con uno más claro. Solo se reemplaza el cuerpo del método con el nuevo algoritmo. • **Solución:** Solo se reemplaza el cuerpo del método nuevo con el nuevo algoritmo.
Move method	• **Razón de refactorización:** Un método es o será usado por mas características de otra clase que la clase en la que está definido. Moviendo métodos, se puede simplificar las clases y terminar siendo una implementación más nítida de un conjunto de responsabilidades. • **Señales de aplicación:** Cuando las clases tienen demasiado

	comportamiento o cuando las clases están colaborando demasiado y están demasiado y están demasiado acopladas. Se tienen métodos que parecen hacer más referencia a otro objeto que al objeto en el que vive. • **Solución:** Crear un nuevo método con un cuerpo similar en la clase que más utiliza. Convierta el método anterior en una simple delegación o elimínelo por completo.
Move Field	• **Razón de refactorización:** Un atributo es, o será, usado por otra clase más que la clase en la que está definido. • **Señales de aplicación:** Cuando más métodos de otra clase usan el atributo más que en la clase en la que se encuentra. • **Solución:** Crea un nuevo atributo en la clase de destino y cambia todos sus clientes.
Extract class	• **Razón de refactorización:** Se tiene una clase haciendo un trabajo que deberían hacer dos. • **Señales de aplicación:** Una clase es demasiado grande para entenderla fácilmente. Una buena señal es que un subconjunto de los datos y un subconjunto de los métodos parecen ir de la mano. Otros buenos signos son subconjuntos de datos que generalmente cambian juntos o dependen particularmente el uno del otro. • **Solución:** Crear una nueva clase y mover los campos y métodos relevantes de la clase origen a la nueva clase.
Inline class	• **Razón de refactorización:** Se tiene una clase que no tiene mucha funcionalidad, es el opuesto de extract class.

	• **Solución:** Mover todas las características a otra clase y eliminar la clase origen
Hide delegate	• **Razón de refactorización:** Un cliente está llamando a una clase delegada de un objeto. • **Señales de aplicación:** La encapsulación significa que los objetos necesitan saber menos sobre otras partes del sistema. • **Solución:** Crear métodos en el servidor para ocultar la delegación.
Remove Middle Man	• **Razón de refactorización:** Una clase está haciendo demasiada delegación simple • **Señales de aplicación:** Entre la motivación de usar Hide Delegate, se tiene las ventajas encapsular el uso de un objeto delegado, sin embargo, el precio que se paga por esto es que cada vez que un cliente quiera usar una nueva función del delegado, debe agregar un método de delegación simple al servidor. Después de agregar características por un tiempo, se vuelve costoso. La clase servidor es solo un intermediario, y tal vez es hora de que el cliente llame al delegado directamente. • **Solución:** Hacer que el cliente llame al delegado directamente.
Introduce Foreign Method	• **Razón de refactorización:** Una clase servidor que se está utilizando necesita un método adicional, pero no se puede modificar la clase. • **Solución:** Crea un método en la clase cliente con una instancia de la clase servidor como primer argumento.
Introduce Local Extension	• **Razón de refactorización:** Una clase servidor que se está utilizando necesita varios métodos

	adicionales, pero no se puede modificar la clase. • **Señales de aplicación:** Se requiere crear muchos métodos foráneos en una clase servidor, o se tienen muchas clases que necesitan el mismo método foráneo. • **Solución:** Crear una nueva clase que contenga estos métodos adicionales, ya sea convirtiendo esta clase de extensión en una subclase o en una envoltura "wraper" de la clase original.
Self Encapsulate Field	• **Razón de refactorización:** Se está accediendo a un atributo directamente, pero el acoplamiento al atributo está siendo incómodo • **Señales de aplicación:** Cuando se trata de acceder a los atributos, hay dos escuelas de pensamiento. Una que sostiene que de la clase donde se define la variable, debe acceder a la variable libremente (acceso directo a la variable). Otra afirma que incluso dentro de la clase, siempre debe usar los accesos (acceso variable indirecto). • **Solución:** Crear métodos get y set para el atributo y usar solo aquellos para accederlo
Replace Data Value with Object	• **Razón de refactorización:** Se tiene un elemento de datos que necesita datos o comportamiento adicionales. • **Señales de aplicación:** A menudo, en las primeras etapas de desarrollo, se toman decisiones sobre la representación de hechos simples como elementos de datos simples. A medida que avanza el desarrollo, te das cuenta de que esos elementos simples ya no son

	tan simples. Un número de teléfono, por ejemplo. • **Solución:** Convierte el elemento de datos en un objeto.
Replace Array with Object	• **Razón de refactorización:** Se tiene una matriz en la que los elementos son heterogéneos. • **Señales de aplicación:** Las matrices son una estructura común para organizar datos. Sin embargo, deberían usarse solo para contener una colección de objetos similares en algún orden. Las convenciones como “el primer elemento en la matriz es el nombre de la persona” son difíciles de recordar; con un objeto, por el contrario, se pueden usar nombres de atributos y métodos para transmitir esta información para que no tenga que recordarla o esperar a que los comentarios estén actualizados. • **Solución:** Reemplace la matriz con un objeto que tenga un atributo para cada elemento de la matriz.
Duplicate Observed Data	• **Razón de refactorización:** Se tiene datos de dominio disponibles solo en un control GUI, y los métodos de dominio necesitan acceso. • **Señales de aplicación:** Un sistema bien organizado separa el código que maneja la interfaz de usuario del código que maneja la lógica de negocios. Hace esto por varias razones. Es posible que desee varias interfaces para una lógica comercial similar; la interfaz de usuario se vuelve demasiado complicada si hace ambas cosas; es más fácil mantener y desarrollar objetos de dominio separados de la GUI.

	• **Solución:** Copiar los datos a un objeto de dominio. Configurar un observador para sincronizar las dos piezas de datos.
Change Unidirectional Association to Bidirectional	• **Razón de refactorización:** Se tiene dos clases que necesitan usar características de cada uno, pero solo hay un enlace unidireccional. • **Solución:** Agregar punteros de regreso y cambiar los modificadores para actualizar ambos conjuntos
Change Bidirectional Association to Unidirectional	• **Razón de refactorización:** Se tiene una asociación bidireccional entre dos clases, pero una clase no requiere características de la otra. • **Señales de aplicación:** Las asociaciones bidireccionales son útiles, pero tienen un precio. El precio es la complejidad añadida de mantener los enlaces bidireccionales y garantizar que los objetos se creen y eliminen correctamente. Las asociaciones bidireccionales no son naturales para muchos programadores, por lo que a menudo son una fuente de errores. Las asociaciones bidireccionales fuerzan una interdependencia entre las dos clases. Cualquier cambio a una clase puede causar un cambio a otra. Si las clases están en paquetes separados, obtienes una interdependencia entre los paquetes. • **Solución:** Eliminar la asociación no requerida.
Replace Magic Number with Symbolic Constant	• **Razón de refactorización:** Se tiene un número literal con un significado particular. • **Señales de aplicación:** Los números mágicos son uno de los males más antiguos de la

	informática. Son números con valores especiales que generalmente no son obvios. Los números mágicos son realmente desagradables cuando necesita hacer referencia al mismo número lógico en más de un lugar. Si los números alguna vez cambian, hacer el cambio es una pesadilla. Incluso si no hace un cambio, tiene la dificultad de descubrir que está sucediendo. • **Solución:** Crear una constante, darle un nombre significativo y reemplazar el número mágico con ella.
Encapsulate Field	• **Razón de refactorización:** Se tiene un atributo con acceso público • **Señales de aplicación:** Uno de los principios fundamentales de la orientación a objetos es la encapsulación o la ocultación de datos. Esto dice que nunca debe hacer públicos sus datos. Cuando hace públicos los datos, otros objetos pueden cambiar y acceder a los valores de los datos sin que el objeto propietario sepa. Esto separa los datos del comportamiento. • **Solución:** Volver el atributo privado y proveer los métodos de acceso.
Replace Record with Data Class	• **Razón de refactorización:** Se requiere interactuar con una estructura de registro en un entorno de programación tradicional. • **Señales de aplicación:** Las estructuras de registro son una característica común de los entornos de programación. Hay varias razones para incorporarlos a un programa orientado a objetos. Podría estar copiando un

	programa heredado, o podría estar comunicando un registro estructurado con una API de programación tradicional o un registro de base de datos. • **Solución:** Crear una clase para representar el registro. Dar a la clase un campo privado con un método get y un método set para cada elemento de datos.
Encapsulate Collection	• **Razón de refactorización:** Se tiene un método que retorna una colección. • **Señales de aplicación:** A menudo, una clase contiene una colección de instancias. Esta colección puede ser una matriz, lista, conjunto o vector. Tales casos a menudo tienen los métodos get y set para la colección. Sin embargo, las colecciones deben usar un protocolo ligeramente diferente al de otros tipos de datos. El método get no debe devolver el objeto de la colección en si, porque eso permite a los clientes manipular el contenido de la colección sin que la clase propietaria sepa lo que está sucediendo. También revela demasiado a los clientes sobre las estructuras de datos internos del objeto. • **Solución:** Hacer que devuelva una vista de solo lectura y proporcionar métodos de agregar/quitar.
Replace Type Code with Class	• **Razón de refactorización:** Una clase tiene un código de tipo numérico que no afecta su comportamiento. • **Señales de aplicación:** Los códigos de tipo numérico o enumeraciones, con una característica común de los lenguajes basados en C. El uso de

	nombres simbólicos puede ser bastante legibles, sin embargo, se tiene el problema de que el nombre simbólico solo es un alias; el compilador todavía ve el número subyacente. El tipo de compilador verifica usando el número, no el nombre simbólico. Cualquier método que tome el tipo de código como argumento espera un número, y no hay nada que obligue a usar un nombre simbólico. • **Solución:** Reemplazar el código de tipo numérico con una nueva clase.
Replace Type Code with Subclasses	• **Razón de refactorización:** Se tiene un código de tipo inmutable que afecta el comportamiento de la clase. • **Señales de aplicación:** Si se tiene un código de tipo que no afecta el comportamiento, se puede usar “reemplazar código de tipo con clase”. Sin embargo, si el código de tipo afecta el comportamiento, lo mejor que se puede hacer es usar polimorfismo para manejar el comportamiento variante. Esta situación generalmente se muestra por la presencia de sentencias CASE o construcciones if-then-else. • **Solución:** Para que esta refactorización funcione, el código de tipo debe reemplazarse con una estructura de herencia que albergue el comportamiento polimórfico. Dicha estructura de herencia tiene una clase con subclases para código de tipo
Replace Type Code with State/Strategy	• **Razón de refactorización:** Se tiene un código de tipo que afecta el comportamiento de una clase, pero no se puede utilizar subclases.

	• **Señales de aplicación:** Esto es similar a “reemplazar código de tipo con subclases”, pero se puede usar si el código de tipo cambia durante la vida útil del objeto o si otra razón impide la subclasificación. Utilizar el patrón de estado o de estrategia. El patrón de estado y el de estrategia son muy similares, por lo que la refactorización que se use realmente no importa. • **Solución:** Reemplazar el código de tipo con un objeto de estado.
Replace Subclass with Fields	• **Razón de refactorización:** Se tienen subclases que varían solo en los métodos que devuelven datos constantes. • **Señales de aplicación:** Puede crear subclases para agregar características o permitir que varíe el comportamiento. Una forma de comportamiento variante es el método constante. Un método constante es aquel que devuelve un valor codificado. Esto puede ser muy útil en subclases que devuelven valores diferentes para un cliente. Aunque los métodos constantes son útiles, una subclase que consiste solo en métodos constantes no está haciendo lo suficiente para que valga la pena existir. • **Solución:** Cambiar los métodos como miembros de la superclase y eliminar las subclases.
Decompose Conditional	• **Razón de refactorización:** Se tiene una sentencia condicional (if-then-else) complicada. • **Señales de aplicación:** Una de las áreas de complejidad más comunes en un programa radica en la lógica condicional compleja. A medida que escribe código para probar condiciones y hacer varias

	cosas dependiendo de varias condiciones, se termina con un método bastante largo. La longitud de un método es en si mismo un factor que dificulta su lectura, pero las condiciones aumentan la dificultad. El problema generalmente radica en el hecho que el código, tanto en las comprobaciones de condición como en las acciones, le dice lo que sucede pero puede ocultar fácilmente porque sucede. • **Solución:** Extraer métodos de la condición, para las sentencias then y para las sentencias else.
Consolidate Conditional Expression	• **Razón de refactorización:** Se tiene una secuencia de pruebas condicionales con el mismo resultado. • **Señales de aplicación:** Este tipo de refactorización es importante por dos razones. Primero, aclara el código mostrando una sola condición el lugar de varias juntos. La secuencia tiene el mismo efecto, pero se comunica llevando a cabo una secuencia de comprobaciones separadas que simplemente se hacen juntas. La segunda razón para esta refactorización es que a menudo se extrae un método. Extraer una condición es una de las cosas mas útiles que puede hacer para aclarar su código. Reemplaza una sentencia de lo que se está haciendo con el porque se está haciendo. • **Solución:** Combinar las diferentes pruebas condicionales en una sola expresión condicional y extraerla en un método.
Consolidate Duplicate Conditional Fragments	• **Razón de refactorización:** El mismo fragmento de código se

	está repitiendo en todas las ramas de una expresión condicional. • **Señales de aplicación:** A veces encuentras el mismo código ejecutado en todas las ramas de un condicional. En ese caso, debe mover el código fuera del condicional. Esto aclara lo que varía y lo que permanece igual. • **Solución:** Mover el fragmento de código fuera de la expresión condicional.
Remove Control Flag	• **Razón de refactorización:** Se tiene una variable que funciona como una bandera de control para una serie de expresiones booleanas. • **Señales de aplicación:** Tales banderas de control son más problemáticas de lo que valen. Provienen de reglas de programación estructurada que requieren rutinas con un punto de entrada y un punto de salida. Esta es la razón por la cual los lenguajes tienen declaraciones break y continue para salir de un condicional complejo • **Solución:** Utilizar una sentencia break o return.
Replace Nested Conditional with Guard Clauses	• **Razón de refactorización:** Un método tiene un comportamiento condicional que no muestra claro el flujo normal de ejecución. • **Señales de aplicación:** Si esta utilizando una sentencia if-then-else, está dando el mismo peso a la rama if y a la rama else. Esto comunica al lector que las ramas son igualmente probables e importantes. En cambio, la clausula de guardia dice: “Esto es raro, y si sucede, haga algo y salga” • **Solución:** Usar clausulas de guardia para cada paso especial.

Replace Conditional with Polymorphism	• **Razón de refactorización:** Tiene un condicional que elige un comportamiento diferente según el tipo de objeto. • **Señales de aplicación:** La esencia del polimorfismo es que evita escribir explícitamente una sentencia condicional cuando tiene objetos cuyos comportamientos varían según sus tipos. Como resultado, encuentra que las instrucciones switch o if-then-else son mucho menos comunes en un programa orientado a objetos. • **Solución:** Mover cada rama del condicional a un método sobreescrito en una subclase. Hacer el método original abstracto.
Introduce Assertion	• **Razón de refactorización:** Una sección de código supone algo sobre el estado del programa. • **Señales de aplicación:** A menudo, las secciones de código que funcionan solo si ciertas condiciones son verdaderas. Esto puede ser tan simple como un cálculo de raíz cuadrada que funciona solo en un valor de entrada positivo. Tales suposiciones a menudo no se establecen, pero solo se pueden decodificar mirando a través de un algoritmo. A veces las suposiciones se expresan con un comentario. Una mejor técnica es hacer explícita la suposición escribiendo una afirmación. Una aserción es una declaración condicional que se supone que es siempre cierta. La falla de una aserción indica un error del programador. Como tal, las fallas de aserción siempre deben dar como resultado excepciones no verificadas.

	• **Solución:** Hacer explícita la suposición con una afirmación.