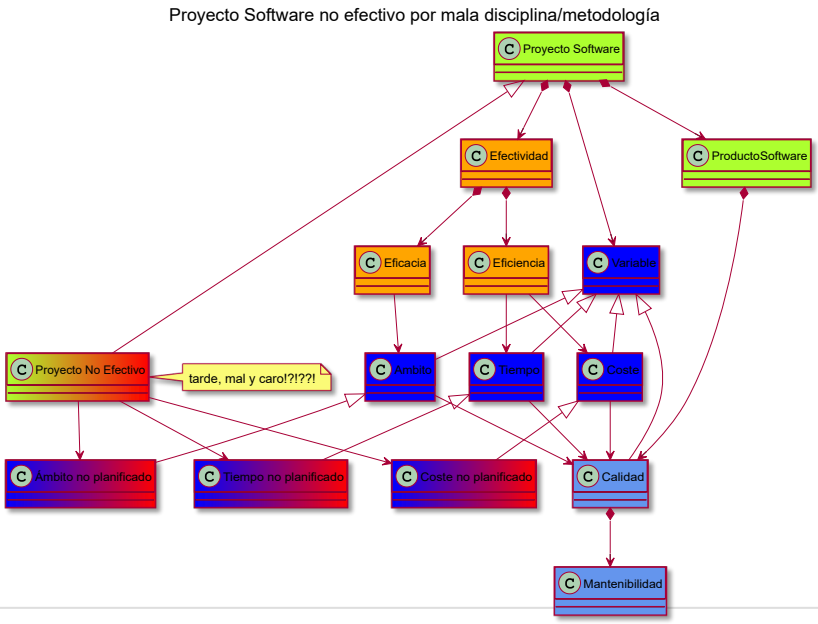
Diseño Modular

Santa Tecla
parqueNaturalSantaTecla@gmail.com
Version 0.0.1

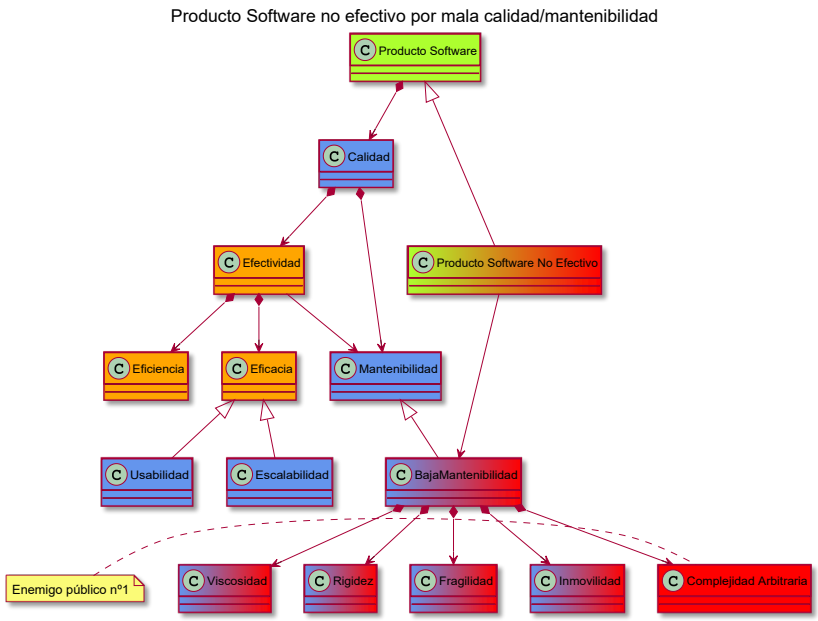
Índice

Justificación: ¿Por qué?

- Proyecto Software poco efectivo
  - porque tiene **malas variables**
    - **tiempo incumplido**,
    - **ámbito incumplido**,
    - **coste incumplido**,
    - *mala calidad*
      - *porque tiene mala mantenibilidad*

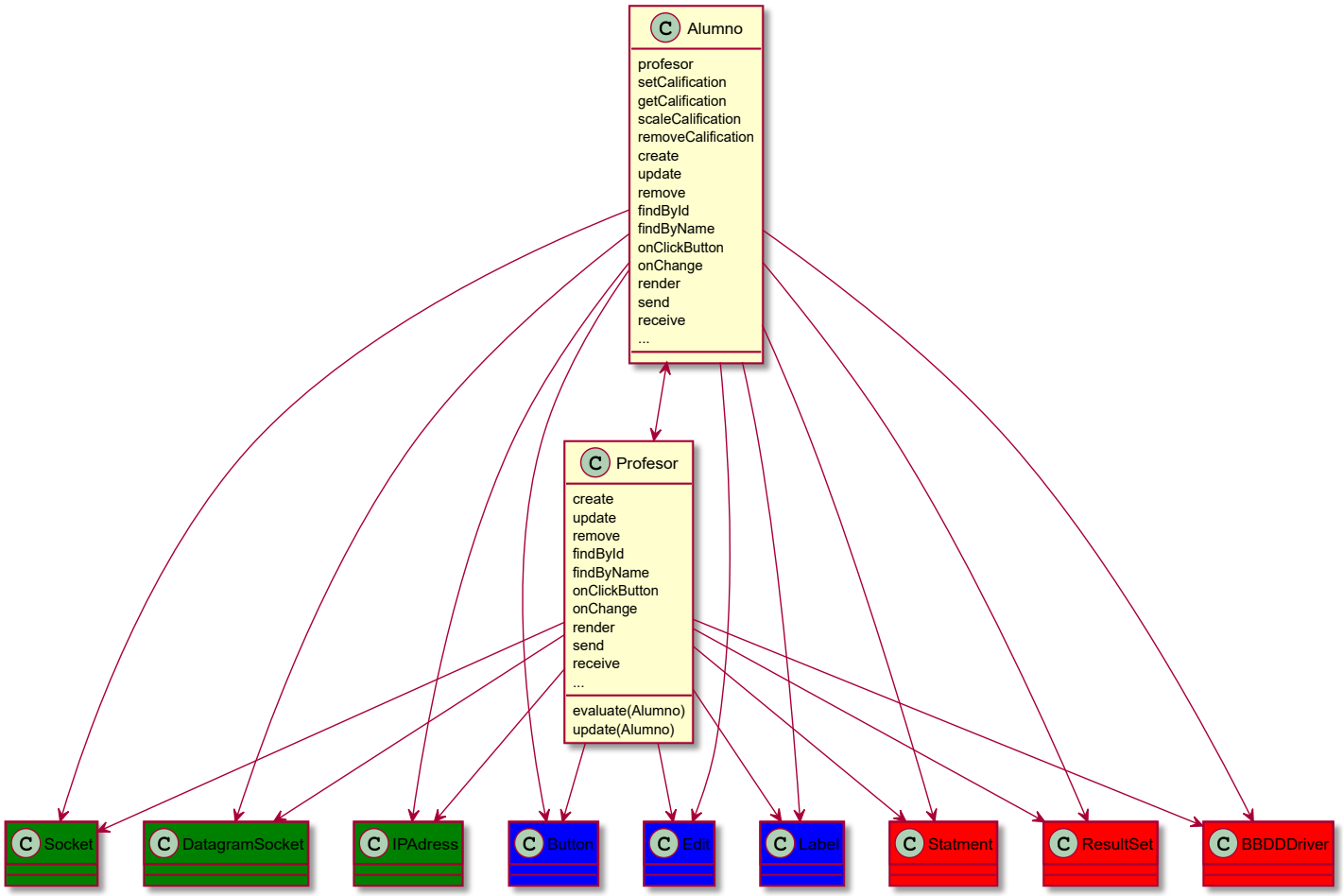


- Producto Software poco efectivo
  - porque tiene **mala calidad**
    - Poco **eficiente**
    - Poco eficaz en **corrección**, **usabilidad**, **escalabilidad**, ...
      - porque tiene **mala mantenibilidad**, de la que depende la eficiencia y la eficacia anteriores, porque es
        - **viscoso**, porque no se puede entender con facilidad
        - **rígido**, porque no se puede cambiar con facilidad
        - **frágil**, porque no se puede probar con facilidad
        - **inmovil**, porque no se puede reutilizar con facilidad



- Hay situaciones en las que una **solución sugerida por Expertos es indeseable**, por lo general a causa de problemas en el **acoplamiento** y **cohesión**.
  - Estos problemas indican violación de un principio básico de diseño: **separación de las principales asuntos** del sistema. El apoyo a una separación de las principales asuntos mejora de acoplamiento y la cohesión en un diseño.
  - Por lo tanto, **a pesar de que por el Experto podría haber alguna justificación para poner la responsabilidad, por otros motivos, por lo general de cohesión y acoplamiento, se trata de un mal diseño**

- *Por ejemplo: una aplicación de gestión educativa con la clase Alumno, que tenga interfaz de texto y gráfica, persistencia en base de datos, comunicaciones, ... Propone la clase Alumno (experto en la información) acoplamiento a tecnologías de interfaz, persistencia y comunicaciones, con la responsabilidad del CRUD de los datos del alumno (alta, baja, modificación y consulta de las notas, datos personales, ...) y mostrarse, persistir y comunicarse porque tiene la información para mostrarse, persistir y comunicarse! Por tanto, muy acoplada a muchas clases, poco cohesiva con muchas responsabilidades y, por tanto, muy grande*
- Por tanto, el Modelo del Dominio con Expertos en la Información es una **inspiránción en el vocabulario del mundo real** pero no imita ni simula ni emula el mundo real.



- El resultado del modelo del dominio con la mejor legibilidad **no asegura un código mantenible, de calidad**:

Viscoso	Presencia de multitud de clases enormes con métodos enormes con acoplamientos cíclicos sin un nítido reparto de responsabiliades
Rigido	Responsabilidades repartidas por multitud de clases que requieren modificaciónes si cambian los requisitos correspondientes
Inmovil	Presencia de multitud de clases acopladas a multitud de clases de tecnologías de diversas tecnologías (GUI, comunicaciones, bases de datos, servicios, ...)
Frágil	Ausencia de red de seguridad de pruebas unitarias por imposibilidad de realizar pruebas sobre las clases anteriores

Definición: ¿Qué?

 escalera

- Se basa en:

Sistemas complejos	• **Jerarquías de módulos** con • **patrones comunes** y con • **separación de asuntos** y • **elementos primitivos relativos** que vienen de un • **sistema anterior** que funcionaba

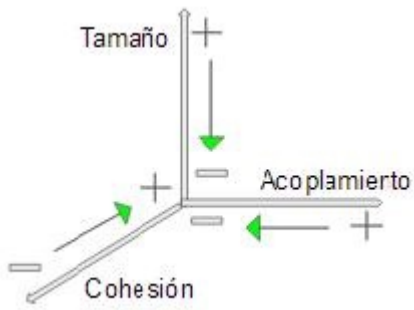
Modelo del Dominio	• Obtener la estructura de **relaciones entre clases** con buenas **abstracciones** e **implementaciones** mediante:
	  ◦ **Análisis del lenguaje**, sustantivos y verbos, cosificación!
	  ◦ **Análisis clásico**, tangibles, intangibles, personas, dispositivos, ...,
	  ◦ **Análisis del dominio**, pero acompañado por un experto
	  ◦ **Diseño por reparto de responsabilidades**, donde cada clase es responsable de lo que tiene que hacer, métodos, y de lo que tiene que conocer, atributos, para hacer lo que tiene que hacer,
	  ◦ **Análisis de casos de uso**, para buscar clases sistemáticamente por cada funcionalidad del sistema

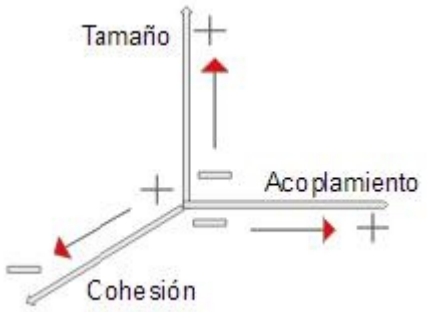
Legibilidad	• Somos **escritores** y respetamos:
	  ◦ **buenos nombres, comentarios y formato**,
	  ◦ **estándares**, consistencias y alarmas, **DRY** y **código muerto**,
	  ◦ **YAGNI**, enfoque, al grano!

- **Diseño Modular** incorpora tres criterios que debe cumplir todo módulo (método, clase y/o paquete):

Alta cohesión	• Única responsabilidad, motivo de cambio, como máximo para cada método, clase y paquete
Bajo acoplamiento	• paquetes con 5 paquetes dependientes máxmio • clases con 5 clases dependientes máximo

Tamaño pequeño	• paquetes con 20 clases máximo • clases con 5 atributos máximo • clases con 20 métodos máximo • métodos con 2 paramétros máximo • métodos con una media de 1,2 parámetros • métodos con 25 líneas como máximo • métodos con 3 niveles de anidación como máximo • métodos con una complejidad algorítmica, número de caminos, de 12 como máximo

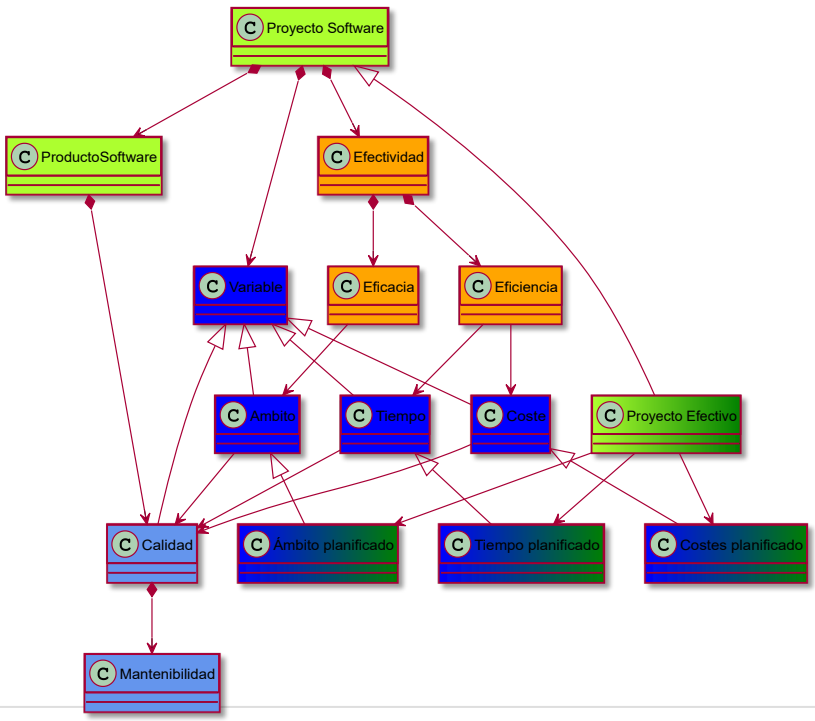




Objetivos: ¿Para qué?

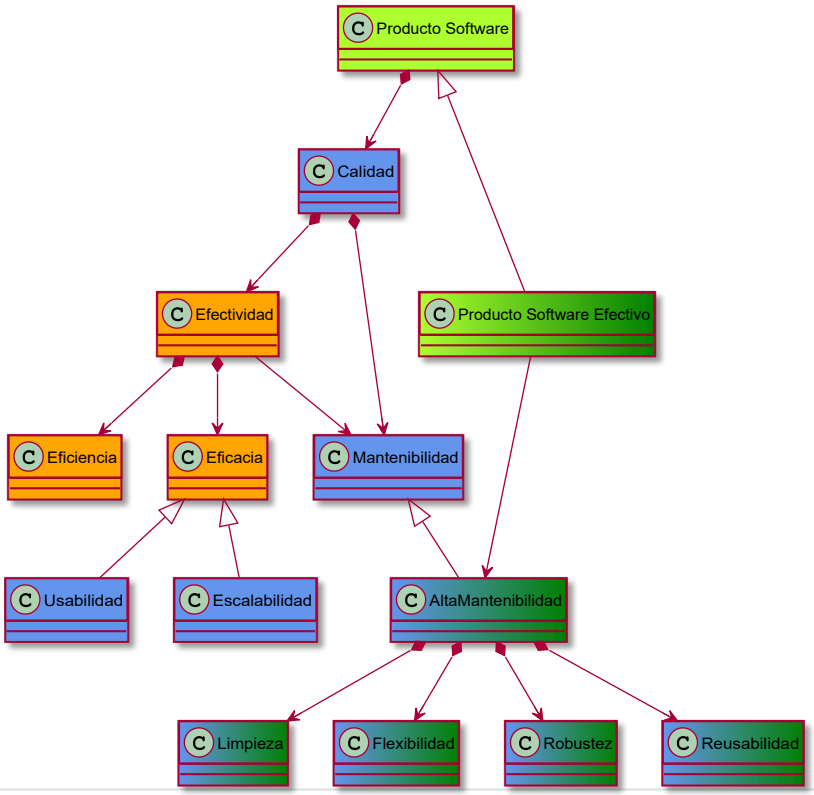
- Proyecto Software efectivo
  - porque tiene **buenas variables**
    - **tiempo cumplido**,
    - **ámbito cumplido**,
    - **coste cumplido**,
    - *buena calidad*
      - *porque tiene buena mantenibilidad*

Proyecto Software efectivo por buena disciplina/metodología



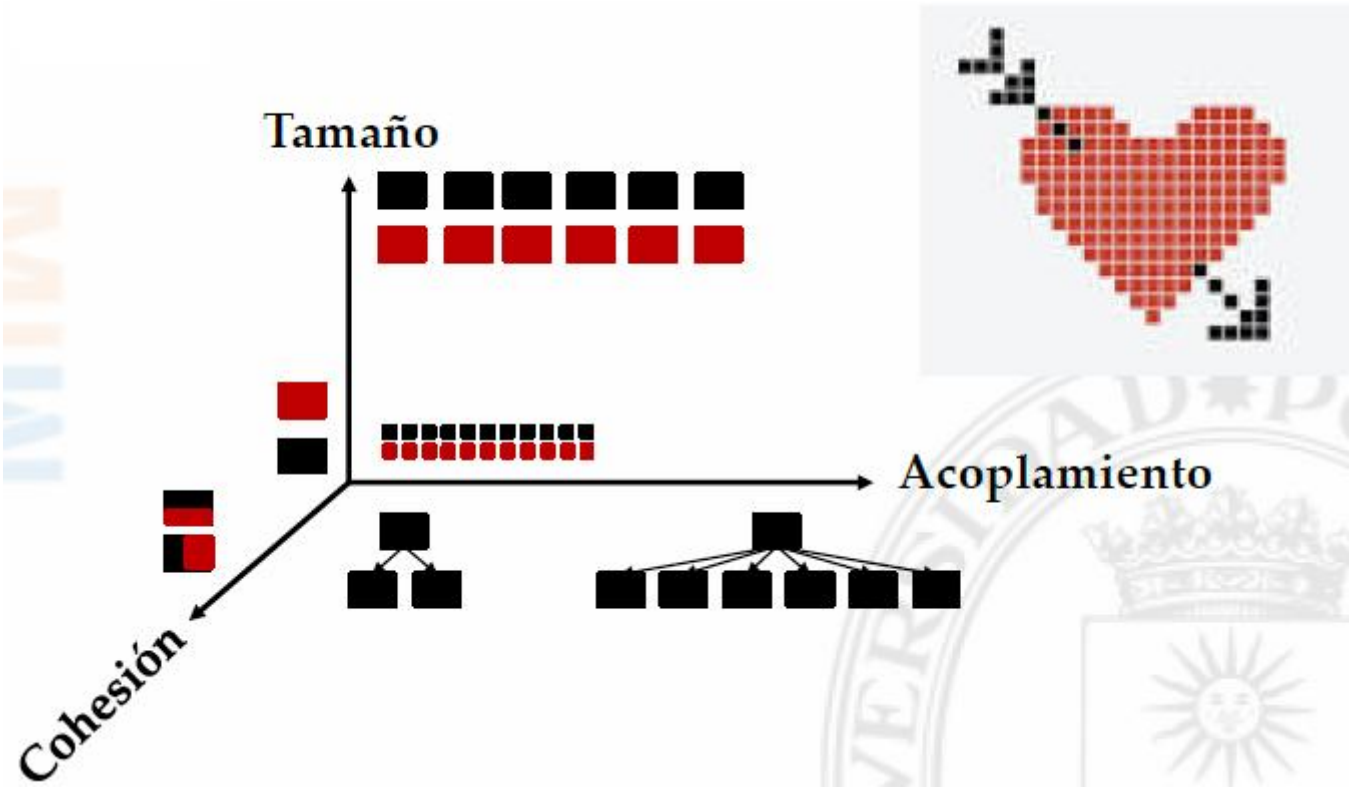
- Producto Software efectivo
  - porque tiene **buena calidad**
    - Es **eficiente**
    - Es eficaz en **corrección**, **usabilidad**, **escalabilidad**, ...
    - porque tiene **buena mantenibilidad**, de la que depende la eficiencia y la eficacia anteriores, porque es
      - **ligero**, porque sí se puede entender con facilidad
      - **flexible**, porque sí se puede cambiar con facilidad
      - **robusto**, porque sí se puede probar con facilidad
      - **reusable**, porque sí se puede reutilizar con facilidad

Producto Software efectivo por buena calidad/mantenibilidad



Fluido	Presencia de multitud de clases pequeñas con métodos pequeños con pequeños acoplamientos acíclicos que puedo recorrer de arriba abajo (top/down o bottom/up), **jerarquía de composición de clases pequeñas, sin ciclos**!

Flexible	Reparto de responsabilidades equilibrado y centralizado en clases que requiere modificarse únicamente si cambian los requisitos correspondientes, **jerarquías de clases con alta cohesión, sin ciclos**!
Resuable	Presencia de multitud de **clases pequeñas, cohesivas y poco acopladas** a tecnologías, algoritmos, ...!
Robusto	Presencia de red de seguridad de pruebas unitarias por posibilidad de realizar pruebas sobre las clases anteriores ... **jerarquía equilibrada de clases pequeñas con alta cohesión y bajo acoplamiento**!



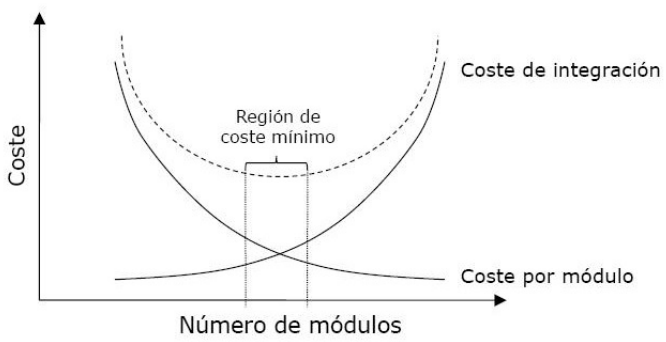
Descripción: ¿Cómo?

Divide y Vencerás

Número de módulos

- Se parte un problema para ser efectivos, eficaces y eficientes, resolviendo problemas más pequeños pero cuando **el problema requiere partirse y no más!**

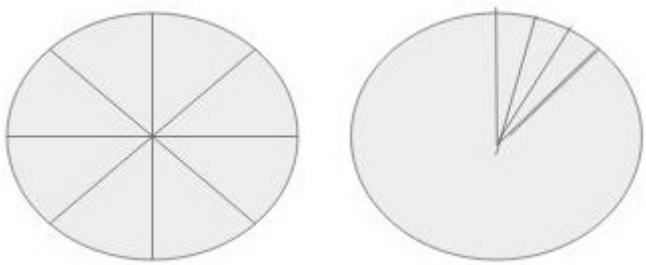
- **Costes de la modularización**, es un compromiso, un **equilibrio**, entre:
  - el **coste de desarrollo** de cada módulo, pocos muy grandes es más costoso que muchos muy pequeños
  - el **coste de integración** de todos los módulos, muy pocos cuesta poco y muchos cuesta mucho



Aplicación mediana (100.000 LOC)	**Muchos módulos**	**Número equilibrado de módulos**	**Pocos módulos**
Costes de Desarrollo	*Reducido*	Equilibrado	**Disparado**
Costes de Integración	**Disparado**	Equilibrado	*Reducido*
Costes Totales	**Disparado**	Equilibrado	**Disparado**

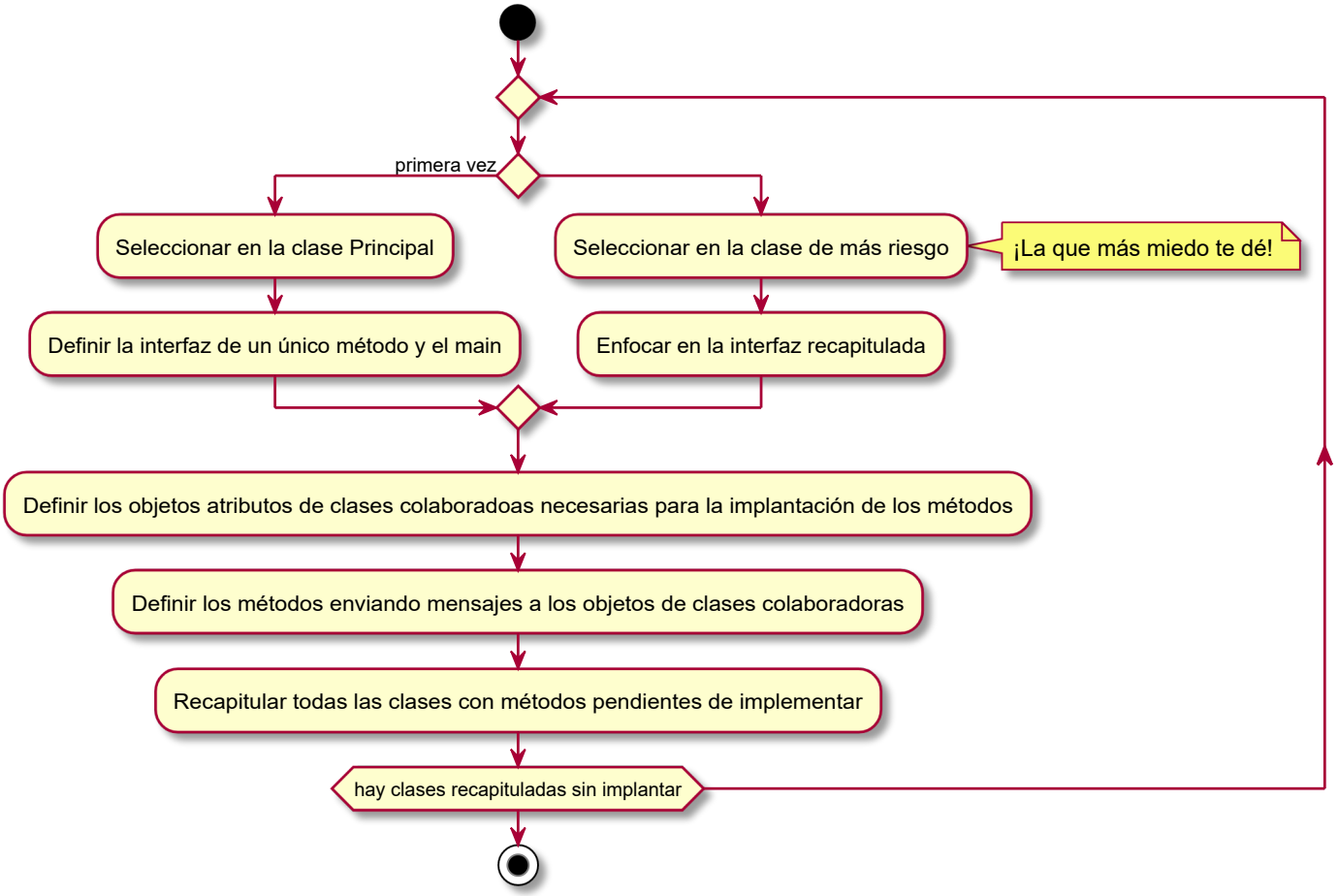
Distribución de Responsabilidades

- **Distribución de Responsabilidades**, también de forma **equilibrada**, con una media y una desviación típica reducidas de la carga relativa de la responsabilidad total
  - **Hay que partir el problema, no trasladarlo!**



Diseño dirigido por Niveles

- **Diseño descendente** (*top/down*);



- **Diseño ascendente** (*bottom/up*):
  - Se comienza con las clases básicas (hojas en las jerarquías) adivinando la responsabilidad necesaria para las superiores
  - Se continúa con clases intermedias (basadas en las hojas de las jerarquías) adivinando la responsabilidad necesaria para las superiores
  - Se repite el paso segundo hasta llegar a la clase principal del sistema
- **Comparativa**

	Diseño descendente	**Diseño ascendente**
Pruebas	Imposible realizar pruebas unitarias hasta llegar a la implantación de las clases hoja de las jerarquías a no ser que se desarrollen **"sustitutos"** (*mocks*) para todas las clases en cada prueba	Se pueden realizar **pruebas unitarias/familiares desde la primera clase**
Reparto de responsabilidades	Muy sencillo **bajo demanda de clases anteriores** pero cualquier error de diseño implica revisar las clases anteriores de la jerarquía de dependencias	Muy complejo **adivinando la responsabilidad**, requiere mucha **experiencia en desarrollo del software y en el dominio de la aplicación**

- En cualquier caso, en cada momento estás en un nivel de la jerarquía, sin ciclos, con un número limitado de elementos

Interfaz

Interfaz Suficiente, Completa y Primitiva

- Por **suficiente**, queremos decir que la clase o módulo captura suficientes características de la abstracción para permitir una interacción significativa y eficiente. De otra manera el componente será inútil. En la práctica, violaciones de esta característica se detectan muy temprano; tales deficiencias se levantan casi **cada vez que construimos un cliente** que debe utilizar esta abstracción.
  - *Por ejemplo: Una clase conjunto de elementos, si ofrece eliminar un elemento deberá contemplar añadir un elemento*
- Por **completo**, nos referimos a que la interfaz de la clase o módulo de captura todas las características significativas de la abstracción. Considerando que la suficiencia implica una interfaz mínima, una interfaz completa es una que **cubre todos los aspectos de la abstracción**. Una clase o módulo completo es, pues, una cuya interfaz es lo suficientemente general como para ser comúnmente utilizable para cualquier cliente. La completitud es una cuestión subjetiva, y puede ser exagerada. Proporcionar todas las operaciones significativas para una abstracción particular, abruma al usuario y en general es innecesaria, ya que muchas operaciones de alto nivel pueden estar compuestas por las de bajo nivel.
  - *Por ejemplo: La clase cadena de caracteres contempla todas y cada una de las operaciones previsibles: esPalíndromo, esEmail, invertir, rimas, ...*
  - Operaciones **primitivas** son aquellas que puede ser **implementadas de manera eficiente sólo si es dado el acceso a la representación subyacente de la abstracción**. Una operación es indiscutiblemente primitiva si podemos implementarla sólo a través del acceso a la representación subyacente. Una operación que podría implementarse sobre las operaciones primitivas existentes, pero a costa de muchos más recursos computacionales, es también un candidato para su inclusión como una operación primitiva.
  - *Por ejemplo: En la clase conjunto de elementos, añadir un elemento es una operación primitiva pero añadir 4 elementos para un cliente particular no sería una operación primitiva porque podría apoyarse eficientemente en la anterior.*

Principios del Menor Compromiso y la Menor Sorpresa

Sinónimos	**Synonyms**	**Libro**	**Autor**
Los nombresde las funciones deberían decirlo que hacen	Function Names Should Say What They Do	Smell Code (Clean Code)	Robert Martin

Antónimos	**Antonyms**	**Libro**	**Autor**
Comportamiento obvio no está implementado	Obvious Behavior Is Unimplemented	Smell Code (Clean Code)	Robert Martin
Responsabilidad fuera de lugar	Misplaced Responsibility	Smell Code (Clean Code)	Robert Martin

> ***Principio del menor compromiso**, a través del cual la interfaz de un objeto proporciona su **comportamiento esencial, y nada más***
>
> *— Abelsony Sussman*

> ***Principio de la menor sorpresa**, a través del cual una abstracción captura todo el **comportamiento de un objeto, ni más ni menos, y no ofrece sorpresas o efectos secundarios** que van más allá del ámbito de la abstracción*
>
> *— Booch*

Código Sucio por Clases Alternativas con Interfaces Diferentes

- **Sinónimos**:

> *Clases Alternativas con Diferentes interfaces (Alternative Classes with Different Interfaces)*
>
> *— Smell Code (Refactoring); Martin Fowler*

- **Justificación**: Complejidad innecesaria
- **Solución**:
  - Renombra los métodos que hacenlo mismo pero tienen nombre diferentes sin la oportuna sobrecarga.
  - Mueve los métodos a clases padre o como poco la interfaz
  - Mueve responsabilidades de las clases hasta que los métodos hacen lo mismo y tienen el mismo nombre: homogenizar el código

Diseño por Contrato

- **La corrección** sólo tiene sentido en relación con una determinada especificación
  - Un **fallo** es cuando un sistema software **se aparta** de su comportamiento especificado durante una de sus ejecuciones
  - Un **defecto** es una propiedad de un sistema de software que pueden hacer que el sistema **se aparte** de su comportamiento especificado.
  - Un **error** es una **mala decision** hecha durante el desarrollo de un sistema software que produce defectos
  - **Un fallo es el hecho real y un defecto es posibilidad potencial**
  - **Los fallos son debidos a los defectos los cuales resultan de los errores**
- **Tiposde error**:
  - **Errores Excepcionales**: producidos por recursos (ficheros, comunicaciones, bibliotecas, ...) fuera del ámbito del software que los maneja.
  - **Errores Lógicos**: producidos por la lógica de un programa que no contempla todos los posibles valores de datos;
  - **Contexto**: ciertos errores (ej.: un valor negativo para calcular un factorial, una referencia sin la dirección de un objeto -null-, ...) pueden ser un **error lógico o excepcional dependiendo del software en el que se está desarrollando**:
    - En el desarrollo de una **aplicación se debe responsabilizar de la detección y subsanación de los errores lógicos dentro de su ámbito en la fase de desarrollo y pruebas**.
    - En el desarrollo de una **biblioteca NO se puede responsabilizar del uso indebido de los servicios prestados a las aplicaciones y NUNCA debe responsabilizarse de la subsanación de dichos errores**. En estos casos, estos errores lógicos se considerarán excepcionales porque la causa del error está **fuera de los límites del software** de la biblioteca.
- **Gestión de Errores**:
  - La robustez, la capacidad del software de reaccionar a casos no incluidos en la especificación, de los posibles errores excepcionales se cubre generalmente con excepciones.
    - *Por ejemplo: abrir un fichero no existente o sobre un soporte dañado, envío y recepción de datos sin conexión en red o con la base de datos, uso inadecuado de una biblioteca, ...*
  - La corrección, la capacidad del software de ejecutar de acuerdo con sus especificaciones, de los posibles errores lógicos se cubre generalmente **inadecuadamente con Programación defensiva** y **adecuadamente con Aserciones**
- **Programación Defensiva**: para obtener software fiable se debe diseñar cada componente de un sistema de modo que se proteja a sí mismo tanto como sea posible.
  - La solución es que cada componente (método) compruebe la viabilidad de operar con *if-then-else*. Pero:
    - **No basta con informar por pantalla** del error lógico porque no se puede acoplar dicho componente a la vista con tecnologías alternativas (consola, gráfica, móvil, web, ...) y porque habrá que avisar al cliente para que tome las medidas oportunas ante el error
    - **No basta con un código de error** cuando no es posible acordar un valor particular de error (*0 ó -1*) si toda la gama es una posible solución
    - En caso optar por la Programción Defensiva **tanto el componente como su cliente aumentarán innecesariamente su complejidad** con sentencias *if-then-else* tanto para confirmar la viablidaddel progreso del componente como para comprobar en todos y cada uno de los clientes la ausencia de error generada por el

componente, lo cual además produce código duplicado.

- **Aserciones**: es una expresión involucrada en algunas entidades del software y establece una propiedad que estas entidades **deben satisfacer en ciertos estados de la ejecución** del programa
  - Es una sentencia del lenguaje que permite comprobar las suposiciones del estado del programa en ejecución. Cada aserción contiene una **expresión lógica** que se supone cierta cuando se ejecute la sentencia. En caso contrario, el **sistema finaliza la ejecución** del programa y **avisa del error detectado**
  - Estas aserciones se pueden usar:
    - **En producción**, para 'documentar formalmente' (compilables) los límites del ámbito del componente sin efecto sobre la ejecución; o
    - **En pre-producción**, para compobraciones automáticas durante la ejecución y, en caso de error, elevar una excepción que termina la ejecución e informa claramente de lo que sucedió
- **Diseño por Contrato** es verlas relaciones entre una clase y sus cientes como un contrato formal expresando los derechos y las obligaciones de cada parte.
  - La vista exterior de cada objeto define un contrato sobre aquellos objetos que pueden depender de él y el cual a su vez debe llevar a cabo en la vista interna del propio objeto, a menudo colaborando con otros. Este contrato establece todas las asunciones que un objeto cliente puede hacer sobre el comportamiento de un objeto servidor.
  - **Objetivos**:
    - Producir software correcto desde el principio porque es diseñado para ser correcto
    - Obtener mucha mejor compresión del problema y sus eventuales soluciones
    - Facilitar la tarea de documentación del software
  - **Protocolo** es el conjunto entero de operaciones que un cliente puede realizar sobre un objeto junto con las "consideraciones legales" en los que pueden ser invocadas.
    - Para cada operación asociada con un objeto, se pueden definir precondiciones y postcondiciones: {P} A {Q}: donde A denota una operación; P y Q son aserciones sobre las propiedades de varias entidades involucradas; P es llamada precondición y Q postcondición.
    - Cualquier ejecución de A, comienza en un estado que cumple P y terminará en un estadoque cumple Q
    - Si la precondición es violada, significa que un cliente no ha satisfecho su parte del contrato y el servidor no puede proceder con fiabilidad.
    - Si una postcondición es violada significa que un servidor no ha llevado a cabo suparte del contrato y sus cliente no pueden confiar en el comportamiento del servidor
    - La pareja precondición/postcondición de una rutina describen el contrato que la rutina(servidor de un cierto servicio) define para sus usuarios (clientes del servicio)
  - **Las Precondiciones** atan al cliente con las restricciones sobre el estado de los parámetros y del objeto servidor que se deben cumplir para una llamada legítima a la operación y que funcione apropiadamente. Son una obligación para el cliente y un beneficio para el servidor.
    - **Precondiciones fuertes exigen más al cliente para solicitar** una tarea y facilitan el trabajo del servidor restringiendo las condiciones de partida
    - **Precondiciones débiles** exigen menos al cliente para solicitar una tarea pero **complican el trabajo del servidor** ante más amplitud en la condiciones de partida
  - **Las Postcondiciones** atan al servidor con las restricciones sobre el estado del valor devuelto y del objeto servidor que se deben cumplir tras el retorno de la operación para que el cliente progrese adecuadamente. Son una obligación para el servidor y un beneficio para el cliente:

- **Postcondiciones fuertes exigen más al servidor que debe de cumplir dicha condición** y facilitan al cliente con un resultado más restringido
- **Postcondiciones débiles** exigen menos al servidor que debe de cumplir dicha condición y **complican al cliente con un resultado más abierto**

	Obligacion	**Beneficiario**
Cliente	Satisfacer laS precondiciones	No necesita comprobar valores de salida porque el resultado garantiza el cumplimiento de la postcondicion
Servidor	Satisfacer las postcondiciones	No necesita comprobar los valores de entrada porque la entrada garantiza el cumplimiento de la precondicion

- Una **Invariantede Clase** es una aserción expresada como una restricción general de la consistencia a aplicar a cada objeto de la clase como un todo.
  - Es diferencte de las precondiciones y postcondiciones caracterizadas a rutinas individuales sobre sus parámetros de entrada y sus resultados respectivamente junto con el estado del objeto. La invariante solo involucra el estado del objeto.
  - Añadir Invariantes de Clase fortalece o mantiene como poco las precondiciones y postcondiciones porque la invariante:
    - **Facilita el trabajo del componente** porque además de la precondición, se puede asumir que el estado inicial del objeto cumple la invariante, lo que restringe el conjunto de casos que se deben contemplar
    - **Complica el trabajo del componente** porque además de la postcondición, se debe cumplir que el estado final del objeto cumpla la invariante, lo que puede aumentar las acciones a realizar
  - Una clase es **correcta** asi:
    - **Cada constructor** de la clase, cuando se aplica satisfaciendo su precondición en un estado donde los atributos tienen sus valores por defecto, cuando termina satisface la invariante: *{P} constructor {Q and I}*
    - **Cada operación** de la clase, cuando se aplica satisfaciendo su precondición y su invariante, cuando termina satisface su postcondición y su invariante: *{P and I} operación {Q and I}*

Implemetación

Cohesión

Cohesión de Métodos

Sinónimos	Synonyms	Libro	Autor
La funciones deberían hacer una sola cosa	Functions Should Do One Thing	Smell Code (Refactoring)	Martin Fowler

- **Justificación**:
  - A menudo se intenta crear funciones que tienen multiples secciones que realizan una serie de operaciones. Dicho de otra manera, la relacion entre las líneas de la implantación del método no son cohesivas porque persiguen distintos objetivos
  - Producen un acoplamiento temporal e imposibilitan su reusabilidad.
    - *Por ejemplo: Método que calculala longitude de un Interval y muestra el resultado por pantalla. Cuando solo se necesitael cálculo, no es reutilizable*
  - **Solución**: Deberían ser convertidas varias funciones pequeñas que hacen una sola cosa

Principio de Única Responsabilidad

- Definido por Robert Martin (*Single Responsibility Principle* -SRP) como uno de los principios SOLID
- Está inspirado en los trabajos de *De Marco* y *Page-Jones*, denominado como **cohesion**: relación funcional de los elementos de un modulo. Pero desplaza un poco el significado y relaciona la cohesion con la causa de cambiode un modulo.
  - Define responsabilidad como una razón de cambio: si se puede pensar en más de un motivo de cambio para una clase, entonces la clase tiene más de una responsabilidad.
  - **Principio de Única Responsabilidad** dice que **una clase debería tener un único motivo de cambio**
    - Es uno de los principios más sencillos y uno de los más difíciles de aplicar correctamente. Combinar responsabilidades es algo que hacemos de forma natural. Encontrar y separar esas responsabilidades entre sí es mucho de lo que el diseño de software es en sí mismo realmente.
    - Un eje de cambio es solo un eje de cambio si el cambio ocurre actualmente. No es prudente aplicar el SRP, o cualquier otro principio para el caso, si no hay ningún síntoma: YAGNI
  - **Justificación**:
    - Si una clase tiene más de una responsabilidad entonces pueden llegar a acoplarse.
    - Los cambios de una responsabilidad pueden perjudicar o inhibir la capacidad de otras clases afectando a su funcionalidad
    - Esta clase de acoplamientos produce diseños frágiles que se rompen de forma inesperada.
  - *Ejemplos:*
    - *si una clase Board_es responsable de las fichas de los jugadores y ademásde presentarse por consola, cuando se cambie a un entorno gráfico puede afectar a la clase que crea el tablero porque tiene que suministrarlos aspectos gráficos necesarios para su nueva presentación*
    - *Si una entidad del dominio (Student, ...) se autoguarda en la base de datos puede repercutir al cambiar la ecnologías de la capa de persistencia en aquellas clases que manejan la entidad*

- **Solución**: Partirla funcionalidad en dos clases. Cada clase maneja un única responsabilidad y en el futuro, si se necesita realizar algún cambio se realizará en la clase que lo maneje.
  - *Ejemplos:*
    - *Separarla clase Board responsable de la gestión de las fichas de los jugadores de la clase BoardView responsable de su visualización colaborando con la clase anterior para obtener la información a presentar. Los cambios en las tecnologías de visualización afectarán únicamente a las clases de presentación*
    - *Separa la clase de entidad del dominio de las clases dedicadas a la grabación y recuperación de dicha entidad(patron DAO)*

Código Sucio por Cambios Divergentes

Sinónimos	**Synonyms**	**Libro**	**Autor**
Cambio divergente	Divergent Change	Smell Code (Refactoring)	Martin Fowler

- **Justificación**: Ocurre cuando una clase se cambia frecuentemente de diferentes maneras, por diferentes razones. Si nos fijamos en una clase y dice: "Bueno, voy a tener que cambiar estos tres métodos cada vez que tengo una nueva base de datos, tengo que cambiar estos cuatro métodos cada vez que hay un nuevo instrumento financiero, ..."
- **Solución**:
  - Es probable que tenga una situación en la que varios objetos son mejor que uno. De esta manera cada objeto sólo se cambia como resultado de un tipo de cambio. Por supuesto, a menudo se descubre esto sólo después de añadir un par de bases de datos o instrumentos financieros.
  - Estructuramos nuestro software para hacer el cambio más fácil. Después de todo, el software está destinado a ser blando. Cuando hacemos un cambio queremos la ventaja de ser capaces de saltar a un solo punto en el sistema y hacer el cambio.

Código Sucio por Cirugía a Escopetazos

Sinónimos	**Synonyms**	**Libro**	**Autor**
Cirugía de escopeta	Shotgun Surgery	Smell Code ((Refactoring)	Martin Fowler

- **Justificación**:
  - Cuando cada vez que se hace una especie de cambio, lo que se tiene que hacer es un montón de pequeños cambios en un montón de clases diferentes. Cuando los cambios son por todos lados son difíciles de encontrar y es fácil pasar por alto un cambio importante.
  - Un cambio que altera muchas clases. Idealmente, existe una relación de uno a uno entre los cambios comunes y las clases.
- **Solución**: En este caso, hay que mover las responsabilidades entre las clases para evitarlo. Si no hay una clase actual que parezca una buena candidata, cree una.

Código Sucio por Clase de Datos

Sinónimos	**Synonyms**	**Libro**	**Autor**
Clase de datos	Data Class	Smell Code (Refactoring)	Martin Fowler

- **Justificación**:

- Hay clases que tienen atributos, métodos *get/set* y nada más. Estas clases son soportes de datos tontos y es casi seguro que se manipulan con demasiado detalle por otras clases.
- Las clases necesitan tomar alguna responsabilidad

- **Solución**:
  - Buscar des de dónde se llaman los métodos *get/set* que son usados por otras clases. Intentar mover el comportamiento dentro de la clase de datos.
  - Después eliminar los métodos _get/set_innecesarios

Código Sucio por Envidia de Características

Sinónimos	Synonyms	Libro	Autor
Características de la envidia	Features Envy	Smell Code (Refactoring)	Martin Fowler

- **Justificación**: Un mal olor clásico es un método que parece más interesado en una clase distinta de la que realmente es. El enfoque más común de la envidia son los datos. Multitud de veces se ve un método que invoca media docena de métodos para conseguir calcular un valor de otro objeto.
- **Solución**:
  - El método claramente quiere estar en otro lugar. A veces sólo una parte del método adolece de envidia; en ese caso, extraer el método ponerlo en la clase adecuada.
  - La clave de los objetos es una técnica para empaquetar datos con los procesos utilizados en esos datos.
  - Si se extrae información de objetos de varias clases combinadamente, colocar el método en la clase que más atributos aporta para el cálculo

Código Sucio por Clases Perezosas

Sinónimos	Synonyms	Libro	Autor
Lazy Class	Clase perezosa	Smell Code (Refactoring)	Martin Fowler

- **Justificación**:
  - Cada clase que se crea cuesta dinero para mantenerla y entenderla.
  - Una clase que no está haciendo lo suficiente para justificar el coste por sí mismo debería ser eliminada.
    - A menudo, esto podría ser una clase que paga por su bagaje y se ha reducido con la refactorización.
    - O podría ser una clase que fue añadida a causa de los cambios que estaban previstos, pero nunca llegaron.
- **Solución**:
  - De cualquier manera, dejar que la clase muera con dignidad asignando su escasa responsabilidad a otra clase
  - Si hay subclases que no están haciendo lo suficiente, trate de contraerla jerarquía.

Código Sucio por Obsesión por Tipos Primitivos

Sinónimos	Synonyms	Libro	Autor
Obsesión primitiva	Primitive Obssesion	Smell Code (Refactoring)	Martin Fowler

- **Justificaión**: Las nuevos programadores orientados a objetos, por lo general, son reacios a utilizar objetos pequeños para pequeñas tareas, como la clase Dinero que combinan cantidad y moneda, clase Intervalo con límite superior e inferior y clases especiales de cadenas de caracteres como números de teléfono y códigos postales.
- **Solución**:
  - Puede reemplazar un valor de tipo primitivo por una clase en incorporar su responsabilidad
  - En el caso de que no exista dicha responsabilidad asignable, puede crear un enumerado

Código Sucio por Grupo de Datos

Sinónimos	**Synonyms**	**Libro**	**Autor**
Grupos de datos	Data Clumps	Smell Code (Refactoring)	Martin Fowler

- **Justificación**: Si se encuentran los mismos dos, tres o cuatro elementos de datos juntos en muchos lugares: atributos en un par de clases, los parámetros de muchas cabeceras de métodos.
- **Solución**: **Los grupos de datos que se presentan juntos realmente deben componer su propio objeto.
  - Ante la duda, una buena comprobación sería preguntarse si quitando uno del grupo, ¿los demás tendrían sentido? Si la respuesta es no, forman un grupo de datos

Acoplamiento

Leyes de Demeter

Sinónimos	Synonyms	Libro	Autor
No hablescon extraños	Do not talk to strangers	xxx	Lieberherr
Cadena de Mensajes	Chain of Message	Smell Code (Refactoring)	Martin Fowler

- **Justificación**: Controlar el bajo acoplamiento restringieno a qué objetos enviar mensajes desde un método
- **Solución**:
  - Enviar únicamente a:
    - This
    - Parámetro
    - Atributos
    - Local
  - No enviar **nunca a otros objetos indirectos obtenidos como resultado de un mensaje a un objeto de conocimiento directo**.

Código Sucio por Librería Incompleta

Sinónimos	**Synonyms**	**Libro**	**Autor**
Clase de biblioteca incompleta	Incomplete Library Class	Smell Code ((Refactoring)	Martin Fowler

- **Justificación**:

- Los desarrolladores de clases de biblioteca son raramente omniscientes. No los culpamos por eso, después de todo, rara vez podemos imaginar un diseño hasta su mayoría que hemos construido, así que los desarrolladores de la biblioteca tienen un trabajo muy duro.
- El problema es que a menudo es de mala educación, y por lo general imposible, modificar una clase de biblioteca para hacer algo que te gustaría que hiciera.

- **Solúción**: Crea una clase con los métodos extra adecuados a tus necesidades

Inapropiada Intimidad

- **Justificación**: Una relación bi-direccional complica el desarrollo, las pruebas, la legibilidad, ...
- **Solucíón**:
  - La sobre-intimidad necesita ser rota:
    - Debes arreglar relaciones bidireccionales por unidireccionales
    - Mueve métodos y atributos para separar las piezas que reduzcan la intimidad
    - Si las clases tienen interes es encomún, extrae en una nueva clase poniendolo común a salvo y haz que las demás sean honesta sobre ella.
    - La herencia a menudo puede conducer a la sobre-intimidad. Las subclases van a conocer más de sus padres de lo que a sus padres les gustaría que ellos conocieran. Se puede sustituir por Delegación

> *Algunas clases llegan a alcanzar demasiada intimidad y gastan mucho tiempo ahondando en las partes priva de otras clases.Nosotros pensamos que nuestras clases deberían ser estrictas con reglas puritanas*
>
> — *Fowler*
> *Refactoring. 1999*

Tamaño

Código Sucio por Listas de Parámetros Largas

Sinónimos	Synonyms	Libro	Autor
Lista de Parámetros Larga	Long Parameter List	Smell Code (Refactoring)	Martin Fowler
Demasiados Argumentos	Too Many Arguments	Smell Code (Clean Code)	Robert Martin

- **Justificación**:
  - Son difíciles de entender
  - Son difíciles de probar todas la combinaciones de argumentos
- **Solución**:
  - Eliminar el parámetro cuando puedes obtenerlo a partir de algún objeto que ya conoces
  - Eliminar varios parámetros suministrando un objeto que los facilite
  - Crear un objeto que agrupe varios parámetros y asigna responsabilidad a sus clase
- **Métrica**:
  - Funciones deberán tener un número pequeño de argumentos. Sin argumentos es lo mejor, seguido por uno, dos.Tres debería evitarse y más de tres es muy cuestionable y debe considerarse como un prejuicio.

Código Sucio por Métodos Largos

Sinónimos	Synonyms	Libro	Autor
Métodos largos	Long Method	Smell Code (Refactoring)	Martin Fowler

- **Justificación**:
  - Desde los principios de la programación, los programadores se han dado cuenta de que cuanto más largo es un procedimiento, más difícil es de entender. Los viejoslenguajes conllevaban una sobrecarga en las llamadas a subrrutinas, de tal forma que se persuadía de escribir métodos pequeños.
- **Solución**:
  - El 99% de las veces, se tiene que acortar un método extrayendo otro. Buscar una parte del método que parezca ir bien junta y hacerlo un nuevo método
  - Una buena técnica es mirarlos comentarios o líneas en blanco para separar partes. Son señales de esta clasede distancia semántica. Un bloque de código con un comentario dice que debes reemplazar el bloque con un método cuyo nombre está basado en el comentario
- **Métrica**:
  - Número de Lineas: [10, 15] como máximo
  - Caracteres por línea: [80,120] como máximo
  - Complejidad ciclomática: [10-15] como máximo
- **Implicaciones**:
  - Los nuevos programadores orientados a objetos a menudo sienten que la computación no se hace en ninguna parte, que los programas son secuencias sin fin de delegación. Cuando has vivido con un programa como tal por unos años, aprendes cómo de valorable son todos esos pequeños métodos. Todos los costes de indirección–explicación, compartición y selección –son respaldadas por pequeños métodos

Código Sucio por Clases Grandes

sinónimos	synonyms	libro	autor
Objeto gigante	Big Large Object(BLOB)	Antipatrónde Desarrollo	William H.Brown et al
Demasiada información	Too much Information	Smell Code (Clean Code)	Robert Martin (Uncle Bob)
Large Class	Clase grande	Smell Code (Refactoring)	Martin Fowler

- **Justificación**:
  - Cuando una clase está tratando de hacer demasiado, a menudo aparece con demasiadas variables de instancia. En tal caso, el código duplicado no puede estar muy lejos.
  - Los archivos pequeños son generalmente más fáciles de entender que archivos de gran tamaño.
- **Métrica**:
  - Parece ser posible construir sistemas significativos con archivos que son típicamente de 200 líneas de largo, con un límite máximo de 500. A pesar de que esto no debería ser una regla dura y rápida, debe considerarse muy deseable.
  - 3 atributos de media; 5 como máximo
  - 20 métodos como máximo

- **Solución**:
  - Descomponer la clase otorgando grupos de atributos relacionados a otras clases
  - Si es una clase de interfaz separa los datos y cálculos del dominio en una clase de entidad

Código Sucio por Atributos Temporales

Sinónimos	**Synonyms**	**Libro**	**Autor**
Campos temporales	Temporary Fields	Smell Code (Refactoring)	Martin Fowler

- **Justificación**:
  - A veces se ve un objeto en el que una variable de instancia se establece sólo en ciertas circunstancias. Tal código es difícil de comprender porque tu esperas que un objetos necesite todas sus variables. Tratar de entender por qué una variable está allí cuando no parece ser usada puede crear complejidad innecesaria.
  - Un caso común de atributo temporal se produce cuando un algoritmo complicado necesita varias variables. Debido a que el ejecutor no quería pasar una lista de parámetros enorme, se ponen en atributos. Pero los atributos son válidas sólo durante el algoritmo; en otros contextos son simplemente confusos.
- **Solución**: En este caso, se puede extraer en una clase los atributos y los métodos que lo requieran. El nuevo objeto es un *objeto método* [Beck]