POLITÉCNICA
"Ingeniamos el futuro"

CAMPUS
DE EXCELENCIA
INTERNACIONAL

MiW

Visión General de la Ingeniería Web.
5. Diseño

Luis Fernández Muñoz
https://www.linkedin.com/in/luisfernandezmunyoz
setillofm@gmail.com
http://blogs.upm.es/garabatossoftware
https://twitter.com/garabatSoftware

INDICE

1. Introducción

1. Respetar los estándares de Código Limpio
2. No ser inconsistentes de Código Limpio
3. Atender alertas de Código Limpio

5.1.1. Respetar los estándares de Código Limpio

- Sinónimos:
  - ***Follow Standard Conventions*** [*Clean Code* - **Robert Martin**]
- Justificación:
  - Cada equipo debe seguir un estándar de codificación basado en normas comunes de la industria. Este estándar de codificación debe especificar cosas como dónde declarar variables de instancia; cómo nombrar las clases, métodos y variables; dónde poner los paréntesis; ...
  - Todo el equipo debe seguir estas convenciones. Esto significa que cada miembro del equipo debe ser lo suficientemente maduros como para darse cuenta de que no importa un ápice donde pones tus llaves, siempre y cuando todos estén de acuerdo en dónde ponerlos.
  - El equipo no debe necesitar un documento para describir estos convenios porque su código proporciona los ejemplos.

5.1.2. No ser inconsistente de Código Limpio

- Sinónimos:
  - ***Inconsistency*** [*Clean Code* - **Robert Martin]**
- Justificación:
  - Si haces algo de cierta manera, haz todas las cosas similares de la misma forma.
  - Tenga cuidado con los convenios a los que decide, y una vez elegido, tenga cuidado de seguir a seguirlos.
  - Una simple consistencia como esta, cuando se aplica de forma fiable, se puede conseguir código más fácil de leer y modificar.
    - Si dentro de una función en particular se utiliza una variable *interval*, a continuación, utilizar el mismo nombre de variable en las otras funciones que utilizan objetos *interval*.
    - Si se nombra un método *processVerificationRequest* , a continuación, utilizar un nombre similar, como *processDeletionRequest*, para los métodos que procesan otros tipos de solicitudes.
  - Esto se remonta al Principio de la Menor Sorpresa

5.1.3. Atender alertas de Código Limpio

- Sinónimos:
  - ***Overridden Safeties*** [*Clean Code* - **Robert Martin**]
- Justificación:
  - *El desastre de Chernobyl se debió a que el gerente de la planta hizo caso omiso de cada uno de los mecanismos de seguridad de uno en uno. Los dispositivos de seguridad estaban siendo inconvenientes para ejecutar un experimento. El resultado fue que el experimento no consiguió realizarse y el mundo vio su primera gran catástrofe civil nuclear.*
  - Es arriesgado anular las advertencias.
    - Desactivar ciertas advertencias del compilador (o todas las advertencias!) puede ayudarle a obtener la sensación de tener éxito pero con el riesgo de sesiones de depuración sin fin.
    - Desactivar las pruebas que fallan y decirte a ti mismo que ya conseguiré que pasen más adelante es tan malo como fingir que tus tarjetas de crédito son dinero gratis.

2. Diseño General

1. Formato de Código Limpio
2. Comentarios de Código Limpio
3. Nombrado de Código Limpio
4. Antipatrón "Descomposición Funcional"
5. Antipatrón "Código Muerto"
6. Principio "No vas a necesitarlo"
7. Principio "No te repitas"
8. Principio "Mantenlo sencillo, estúpido!"

5.2.1. Formato de Código Limpio

- Justificación:
  - Formateo de código es importante. Es demasiado importante como para ignorarlo y es demasiado importante como para tratarlo religiosamente. El formateo de código trata sobre la comunicación y la comunicación es de primer orden para los desarrolladores profesionales
- Implicaciones:
  - Una línea entre grupos lógicos (atributos y cada método).
  - Los atributos deben declararse al principio de la clase
  - Las funciones dependientes en que una llama a otra, deberían estar verticalmente cerca: primero la llamante y luego la llamada
  - Grupos de funciones que realizan operaciones parecidas, deberían permanencer juntas
  - Las variables deberían declararse tan cerca como sea posible de su utilización

5.2.1. Formato de Código Limpio

- Implicaciones:
  - Los programadores prefieren líneas cortas (~40, máximo 80/120)
  - Utilizamos el espacio en blanco horizontal para asociar las cosas que están fuertemente relacionadas y disociar las cosas que están más débilmente relacionadas y para acentuar la precedencia de operadores
  - Un código es una jerarquía. Hay información que pertenece al archivo como un todo, a las clases individuales dentro del archivo, a los métodos dentro de las clases, a los bloques dentro de los métodos, y de forma recursiva a los bloques dentro de los bloques. Cada nivel de esta jerarquía es un ámbito en el que los nombres pueden ser declaradas y en la que las declaraciones y sentencias ejecutables se interpretan. Para hacer esta jerarquía visible, hay que sangrar la líneas de código fuente de forma proporcional a su posición en la jerarquía.

5.2.1. Formato de Código Limpio

- Violaciones:
  - No uses tabuladroes entre los tipos y las variables para una disposición por columnas
  - Nunca rompas las reglas de sangrado por muy pequeñas que sean las líneas
- Máximas:
  - "*Conceptos que están estrechamente relacionados deben mantenerse verticalmente cercanos*" [**Martin, R.**]
  - Un equipo de desarrolladores deben ponerse de acuerdo sobre un único estilo de formato y luego todos los miembros de ese equipo debe usar ese estilo.

5.2.2. Comentarios de Código Limpio

- Justificación:
  - Nada puede ser **tan útil como un comentario bien colocado**.
- Implicaciones:
  - Código claro y expresivo con algunos pocos comentarios es muy superior al código desordenado y complejo con un montón de comentarios. En muchos casos es simplemente una cuestión de crear una función con el nombre que diga lo mismo que el comentario.
  - Comentario legal. Ej.: copyright, license, …
  - Comentario aclarativo. Ej.:

```
// format matched kk:mm:ss EEE, MMM dd, yyyy
String format = "\\d*:\\d*:\\d* \\w*, \\w* \\d*, \\d*";
```

5.2.2. Comentarios de Código Limpio

- Violaciones:
  - Nada puede **estorbar más encima de un módulo que frívolos comentarios dogmáticos**.
  - Es simplemente una tontería tener una regla que dice que cada variable debe tener un comentario o que cada función debe tener un *javadoc* a no ser que sea publicado como biblioteca
  - Comentarios redundantes. Ej.: *int dayOfMonth; //the day of the month,*
  - Comentarios de atribución. Ej. *// Added by Luis* para eso está el control de versiones cuando haga realmente falta
  - Comentarios confusos. Si nuestro único recurso es examinar el código en otras partes del sistema para averiguar lo que está pasando.
  - Comentarios inexactos. Un programador hace una declaración en sus comentarios que no es lo suficientemente precisa para ser exacta
  - Comentarios de sección. Ej.: *// Actions ////////////////////////////////*
  - Código comentado. Para eso está el control de versiones

5.2.2. Comentarios de Código Limpio

- Violaciones:
  - Comentarios no mantenidos. Con valores que no se actualizará. Ej.: *// port is 7077*
  - Comentarios excesivos. Como el historial de interesantes discusiones de diseño
  - Nada puede ser **tan perjudicial como un enrevesado comentario desactualizado que propaga mentiras y desinformación**
- Máximas:
  - "*No comentes código malo, reescribelo*" [**Kernighan &. Plaugher**]

5.2.3. Nombrado de Código Limpio

- Sinónimos:
  - Elige nombres descriptivos (***Choose Descriptive Names***) [*Clean Code* - **Robert Martin]**
  - Elige nombres al nivel de abstracción apropiado (***Choose Names at the Appropriate Level of Abstraction***) [*Clean Code* - **Robert Martin]**
  - Usa nomenclatura estándar donde sea posible (***Use Standard Nomenclature Where Possible***) [*Clean Code* - **Robert Martin]**
  - Nombre no ambiguos (***Unambiguous Names***) [*Clean Code* - **Robert Martin]**
  - Usa nombres largo para ámbitos largos (***Use Long Names for Long Scopes***) [*Clean Code* - **Robert Martin]**
  - Evita codificaciones (***Avoid Encodings***) [*Clean Code* - **Robert Martin]**
  - Los nombre debería describir los efectos laterales (***Names Should Describe Side-Effects***) [*Clean Code* - **Robert Martin]**

5.2.3. Nombrado de Código Limpio

- Justificación:
  - **Los nombre deben revelar su intención**. Deberían revelar por qué existe, qué hace, y cómo se utiliza para facilitar la legibilidad para el desarrollo y el mantenimiento correctivo, perfectivo y adaptativo
- Implicaciones:
  - Nombres pronunciables que permitan mantener una conversación
  - Mayúsculas en los cambios de palabra (*CamelCase)*. Ej.:
  - Nombres del dominio del problema y de la solución. Ej.: *Student, discard, …, TicketVisitor, ReaderFactory, …* conocidos por la comunidad de programadores
  - Elige una palabra para un concepto abstracto y aferrarte a él. Ej.: *get, retrieve, fetch, …* es confuso como métodos equivalentes de diferentes clases.

5.2.3. Nombrado de Código Limpio

- Implicaciones:
  - Nombres de paquetes deben ser sustantivos y comenzar en minúsculas. Ej. *models.customers*
  - Nombres de clases deben ser sustantivos y comenzar en mayúsculas. Ej. *Controller*, no *Control*
  - Nombres de métodos deben ser verbos o una frase con verbo y comenzar en minúsculas.
  - Nombres de métodos de acceso deben anteponer *get*(*is* para lógicos) */ set*
- Violaciones:
  - Si un nombre requiere un comentario, el nombre no revela su intención. Ej.: *d, mpd, lista1*, … mejor: *elapsedTimeInDays, daysSinceModificatio, …*
  - Utilizar separadores de palabras como guiones o subrayados

5.2.3. Nombrado de Código Limpio

- Violaciones:
  - Constantes numéricas que son difíciles de localizar y mantener
  - Nombres de una letra y muy en particular, 'O' y 'l' que se confunden con 0 y 1. Excepcionalmente, en variables locales auxiliares de métodos. Un contador de bucle puede ser nombrado *i* o *j* o *k* (pero nunca *l*!) si su alcance es muy pequeño y no hay otros nombres que pueden entrar en conflicto con él. Esto se debe a que esos nombres de una sola letra para contadores de bucles son tradicionales. Es un estándar, "allá donde fueres, haz lo que vieres".
  - Nombres acrónimos a no ser que sean internacionales. Ej.: *BHPS*, … mejor *BehaviourHumanPredictionSystem*
  - Nombres con códigos de tipo o información del ámbito (notación Húngara y similares). Ej.: *int iAge* o *int m_iAge*, … mejor *age; class CStudent,* mejor *Student*

5.2.3. Nombrado de Código Limpio

- Violaciones:
  - Nombre con palabras vacías o redundantes como *Object, Class, Data, Inform, the, a …* Ej.: *StudentData*, *boardObject*, *theMessage*… mejor *Student*, *board*, *message*, …
  - Nombre en serie. Ej. *player1*, *player2*, … mejor *players* o *winnerPlayer* y *looserPlayer*
  - Nombres desinformativos que no son lo que dicen. Ej. *customerList* pero no es una lista, es un conjunto; …
  - Nombres indistinguibles como *XYZControllerForEfficientHandling OfStrings* y *XYZControllerForEfficientStorageOfStrings*
  - Nombres polisémicos en un mismo contexto. Ej.: *book* como registro en un hotel y libro; …
  - Nombres graciosos, juegos de palabras, …

5.2.3. Nombrado de Código Limpio

- Máximas:
  - "*Una línea de código se escribe una vez y se lee cientos de veces*" [**Cox**] La elección de buenos nombres **lleva tiempo, pero ahorra más de lo que toma**.
  - Así que ten cuidado con los nombres y **cámbialos cuando encuentres otros mejores**. Hay personas que tienen miedo de cambiar el nombre de las cosas por temor a que otros desarrolladores objeten. No compartimos que el miedo y encontramos que estamos realmente agradecidos cuando los nombres cambian (para mejor). La mayor parte del tiempo realmente no memorizamos los nombres de clases y métodos. Utilizamos las herramientas modernas para hacer frente a estos detalles como para que podamos centrarnos en si el código se lee como párrafos y oraciones.

5.2.4. Antipatrón "Descomposición Funcional"

- Sinónimos:
  - ***Functional Descomposition*** [Antipatrón de Desarrollo; **William H. Brown** et al]
- Síntomas:
  - Clases con nombres de función
  - Clases con un solo método
  - Ausencia de principios orientados a objetos como herencia, polimorfismo, …
- Justificación:
  - Imposible de comprender el software, de reutilizar, de probar, …
- Solución:
  - Aplicar los Patrones Generals para la Asignacion de Responsabilidades del Software: **GRASP**

5.2.5. Antipatrón "Código Muerto"

- Sinónimos:
  - Antipatrón "***Dead Code***"
  - Flujo de lava "***Lava Flow***"
- Justificación:
  - Según el código muerto se anquilosa y se endurecen, rápidamente se hace imposible documentar el código o entender suficientemente su arquitectura para hacer mejoras.
  - Si no se elimina el código muerto, puede continuar proliferando según se reutiliza código en otras áreas
  - Puede haber crecimiento exponencial según los sucesivos desarrolladores, demasiado apremiados o intimidados por analizar los códigos originales, seguirán produciendo nuevos flujos secundarios en su intento de evitar los originales.

5.2.5. Antipatrón "Código Muerto"

- Síntomas:
  - Fragmentos de código injustificables en el sistema: clases, funciones o segmentos de código complejo no documentado con aspecto importante pero que no están relacionados con el sistema
  - Bloques enteros de código comentado sin explicación o documentación
  - Interfaces no usadas, inexplicables u obsoletas
  - Montones de áreas de código con comentarios: "proceso de cambio", "para ser reemplazado", …

5.2.6. Principio "No vas a necesitarlo"

- Sinónimos:
  - **YAGNI** (*You aren't going to need it* o *You ain't gonna need it*);
  - **Generalidad Especulativa** (*Speculative Generality - Smell Code/Refactoring*)
- Justificación:
  - Siempre se implementan cosas cuando realmente se necesitan, no cuando se prevén que se necesiten. Por tanto, no se debe agregar funcionalidad hasta que se considere estrictamente necesario.
  - Las características innecesarias son inconveniente por:
    - El tiempo gastado se toma para la adición, la prueba o la mejora de funcionalidad innecesaria. Y posteriormente, las nuevas características deben depurarse, documentarse y mantenerse.
    - Conduce a la hinchazón de código y el software se hace más grande y más complicado. Añadir nuevas características puede sugerir otras nuevas características. Si estas nuevas funciones se implementan así, esto podría resultar en un efecto bola de nieve

5.2.6. Principio "No vas a necesitarlo"

- Violaciones:
  - Inconveniente por:
    - Hasta que la característica es realmente necesaria, es difícil definir completamente lo que debe hacer y probarla. Si la nueva característica no está bien definida y probada, puede que no funcione correctamente, incluso si eventualmente se necesitara.
    - A menos que existan especificaciones y algún tipo de control de revisión, la función no puede ser conocida por los programadores que podrían hacer uso de ella.
    - Cualquier nueva característica impone restricciones en lo que se puede hacer en el futuro, por lo que una característica innecesaria puede interrumpir características necesarias que se agreguen en el futuro.
  - Debe ser utilizado en combinación con varias otras prácticas, tales como refactorización continua que implica la continua automatización de pruebas unitarias y la integración continua

5.2.7. Principio "No te repitas"

- Sinónimos:
  - **DRY** (*Don't Repeat Yourself*); Fuente Única de la Verdad;
- Antónimos:
  - Código Duplicado (***Duplicate code***) [*Smell Code - (Refactoring)* **Martin Fowler**]
  - Copiar y Pegar (***Copy+Paste***) [Antipatrón de Desarrollo - **William H. Brown et al**]
  - Duplicación (***Duplication***) [*Smell Code (Clean Code)* **- Robert Martin]**
  - Escribe todo dos veces o disfrutamos tecleando (***write everything twice*** or ***we enjoy typing*** **- WET**);

5.2.7. Principio "No te repitas"

- Justificación:
  - Evitar re-analizar, re-diseñar, re-codificar soluciones que complica el enormemente el mantenimiento correctivo, perfectivo y adaptativo
    - El efecto 2000 paralizó la producción de software y los gobiernos subvencionaron con el dinero de los impuestos a las empresas privadas para reaccionar ante el problema
- Solución:
  - Cada pieza de conocimiento debe tener una única, inequívoca y autoritativa representación en un sistema.
  - El objetivo es reducir la repetición de la información de todo tipo, lo que hace que los sistemas de software sean más fácil de mantener
  - La consecuencia es que una modificación de cualquier elemento individual de un sistema no requiere un cambio en otros elementos lógicamente no relacionados (similar a la 1ªFN de BBDD).
  - Aplicable a la programación, esquemas de bases de datos, planes de prueba, el sistema de construcción, incluso la documentación.

5.2.7. Principio "No te repitas"

- Violaciones:
  - Obviamente, código repetido carácter por carácter con la misma semántica. Cuidado! La misma línea de ámbitos diferentes puede ser diferente código por la declaraciones en las que se apoya.
    - *this.add(element);* puede ser completamente diferente en dos clases
  - Código semánticamente repetido pero con nombres de variables cambiadas, algún orden de sentencias, …
  - Bloque de código que podría sustituirse por llamadas a otros métodos que ya desarrollan esa funcionalidad

5.2.8. Principio "Mantenlo sencillo, estúpido!"

- Sinónimos:
  - Mantenlo sencillo, estúpido! Mantenlo pequeño y sencillo! Mantenlo pequeño y simple! (***Keep it simple, stupid!; Keep it short and simple; or Keep it small and simple*** **- KISS**) [**Kelly Jhonson**]
  - Comprender el algoritmo (***Understand the Algorithm***) [*Smell Code (Clean Code)* **- Robert Martin**]
- Antónimos:
  - Código Espagueti (***Spaghetti Code***) [Antipatrón de Desarrollo - **William H. Brown et al**]
  - Generalidad Espculativa (***Speculative Generality***) [*Smell Code - (Refactoring)* **Martin Fowler**]
  - Intenciones obscuras (***Obscured Intent***) [*Smell Code (Clean Code)* **- Robert Martin**]

5.2.8. Principio "Mantenlo sencillo, estúpido!"

- Justificación:
  - la mayoría de sistemas funcionan mejor si se mantienen simples que si se hacen complejos; por tanto, la simplicidad debe ser un objetivo clave del diseño, y cualquier complejidad innecesaria debe evitarse
  - *En igualdad de condiciones, la explicación más sencilla suele ser la correcta [Navaja de **Occam**]*
  - *Cualquier tonto inteligente puede hacer cosas más grandes y más complejas ... se necesita un toque de genialidad y mucho coraje para moverse en la dirección opuesta" **[Einstein, A.]***
  - *"Sin embargo, no es suficiente para dejar las comillas alrededor de la palabra 'funciona'. Usted debe saber que la solución es correcta. A menudo, la mejor manera de obtener este conocimiento y comprensión es refactorizar la función en algo que es tan limpio y expresivo que es obvio cómo funciona*". **[Martin]**
  - *"La diferencia entre un programador inteligente y un programador profesional es que el profesional entiende que la claridad es el rey. Los profesionales utilizan su potencia para lo bueno y escribir código que otros puedan entender"* **[Martin]**

5.2.8. Principio "Mantenlo sencillo, estúpido!"

- Violaciones:
  - Si tienes clases abstractas que no están haciendo mucho, colapsa la jerarquía
  - Innecesaria delegación puede ser eliminada con la clase "en línea"
  - Métodos que no usan parámetros deberían ser eliminados
  - Nombres de métodos con extraños nombres abstractos deben ser renombrados para "traerlos a la tierra"
  - Complejos algoritmos generalistas para situaciones muy concretas
  - Complejos algritmos muy eficientes cuando no hay necesidad

3. Diseño de Métodos

1. Interfaz Suficiente, Completa y Primitiva
2. Principios del Menor Compromiso y la Menor Sorpresa
3. Cohesión de Métodos
4. Código Sucio por Clases Alternativas con Interfaces Diferentes
5. Código Sucio por Listas de Parámetros Largas
6. Código Sucio por Métodos Largos
7. Código Sucio por Envidia de Características
8. Diseño por Contrato

5.3.1. 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**. Hacer otra cosa hace que el componente sea 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.
  - Una clase conjunto de elementos, si ofrece eliminar un elemento deberá contemplar añadir un elemento

5.3.1. Interfaz Suficiente, Completa y Primitiva

- 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 mínimo, 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.
  - La clase cadena de caracteres contempla todas y cada una de las operaciones previsibles: esPalíndromo, esEmail, …

5.3.1. Interfaz Suficiente, Completa y Primitiva

- 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.
  - 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.

5.3.2. Principios del Menor Compromiso y la Menor Sorpresa

- Sinónimos:
  - Los nombres de las funciones deberían decir lo que hacen (***Function Names Should Say What They Do***) [*Clean Code (Smell Code)* – **Robert Martin**]
- Antónimos:
  - Comportamiento obvio no está implementado (***Obvious Behavior Is Unimplemented***) [*Clean Code (Smell Code)* – **Robert Martin**]
  - Responsabilidad fuera de lugar (***Misplaced Responsibility*** [*Clean Code (Smell Code)* – **Robert Martin**]
- **Principio del menor compromiso**, a través del cual la interfaz de un objeto proporciona su comportamiento esencial, y nada más [**Abelson** y **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**]

5.3.3. Cohesión de Métodos

- Sinónimos:
  - 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.
    - Método que calcula la longitude de un *Interval* y muestra el resultado por pantalla. Cuando solo se necesita el cálculo, no es reutilizable
- Solución:
  - Deberían ser convertidas varias funciones pequeñas que hacen una sola cosa

5.3.4. 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 hacen lo mismo pero tienen nombre diferentes sin la oportuna sobrecarga.
  - Mueve responsabilidades de las clases hasta que los métodos hacen lo mismos y tienen el mismo nombre: homogina el código
  - Mueve los métodos a clases padre o como poco la interfaz

5.3.5. Código Sucio por Listas de Parámetros Largas

- Sinónimos:
  - 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.

5.3.6. Código Sucio por Métodos Largos

- Sinónimos:
  - 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 viejos lenguajes 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 mirar los comentarios o líneas en blanco para separar partes. Son señales de esta clase de 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

5.3.6. Código Sucio por Métodos Largos

- 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

5.3.7. Código Sucio por Envidia de Características

- Sinónimos:
  - ***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

5.3.8. Diseño por Contrato

- La **corrección** sólo tiene sentido en relación con una determinada especificación
  - Un **fallo** es cuando un sistema software se apartar 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 fallo es el hecho real y un defecto es posibilidad potencial
  - Un **error** es una mala decision hecha durante el desarrollo de un sistema software que produce defectos
  - Los fallos son debidos a los defectos los cuales resultan de los errores

5.3.8. Diseño por Contrato

- Tipos de 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.

5.3.8. Diseño por Contrato

- 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.
    - P.e. 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**
    - P.e. cálculo de factorial de un número negativo, bucle infinito, …

5.3.8. Diseño por Contrato

- **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 viablidad del 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.

5.3.8. Diseño por Contrato

- **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

5.3.8. Diseño por Contrato

- **Diseño por Contrato** es ver las 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

5.3.8. Diseño por Contrato

- **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 estado que 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 su parte 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)

5.3.8. Diseño por Contrato

- 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

5.3.8. Diseño por Contrato

- 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

5.3.8. Diseño por Contrato

	Obligación	Beneficio
Cliente	Satisfacer las precondiciones	No se necesita comprobar los valores de salida porque el resultado garantiza el cumplimiento de la postcondición
Servidor	Satisfacer las postcondiciones	No se necesita comprobar los valores de entrada porque la entrada garantiza el cumplimiento de la precondición

5.3.8. Diseño por Contrato

- Una **Invariante de 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

5.3.8. Diseño por Contrato

- Una clase es correcta si:
  - 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** $I\}$

4. Diseño de Clases

1. Código Sucio por Clases Grandes
2. Código Sucio por Clases Perezosas
3. Código Sucio por Obsesión por Tipos Primitivos
4. Código Sucio por Grupo de Datos
5. Código Sucio por Clase de Datos
6. Código Sucio por Librería Incompleta
7. Código Sucio por Cirugía a Escopetazos
8. Código Sucio por Cambios Divergentes
9. Código Sucio por Atributos Temporales
10. Principio de Única Responsabilidad

5.4.1. Código Sucio por Clases Grandes

- Sinónimos:
  - ***Big Large Object – BLOB*** [Antipatrón de Desarrollo; **William H. Brown** et al]
  - ***Too much Information*** [*Smell Code (Clean Code)*; **Robert Martin** (Uncle Bob)]
  - ***Large Class*** [*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.

5.4.1. Código Sucio por Clases Grandes

- 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

5.4.2. Código Sucio por Clases Perezosas

- Sinónimos: ***Lazy Class*** [*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 contraer la jerarquía.

5.4.3. Código Sucio por Obsesión por Tipos Primitivos

- Sinónimos: ***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

5.4.4. Código Sucio por Grupo de Datos

- Sinónimos: ***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

5.4.5. Código Sucio por Clase de Datos

- Sinónimos:
  - ***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 desde 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

5.4.6. Código Sucio por Librería Incompleta

- Sinónimos:
  - ***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

5.4.7. Código Sucio por Cirugía a Escopetazos

- Sinónimos:
  - ***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.

5.4.8. Código Sucio por Cambios Divergentes

- Sinónimos:
  - *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.

5.4.9. Código Sucio por Atributos Temporales

- Sinónimos:
  - *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].

5.4.10. Principio de Única Responsabilidad

- Definido por Robert Martin (*Single Responsibility Principle* - ***SRP***) como uno de los principios SOLID
  - Está inspirado en los trabajos de DeMarco 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 cambio de 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

5.4.10. Principio de Única Responsabilidad

- 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 además de presentarse por consola, cuando se cambie a un entorno gráfico puede afectar a la clase que crea el tablero porque tiene que suministrar los 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 tecnologías de la capa de persistencia en aquellas clases que manejan la entidad

5.4.10. Principio de Única Responsabilidad

- Solución:
  - Partir la 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:
    - Separar la 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)

5. Diseño de Herencias

1. Principio de Sustitución de Liskov
2. Herencia vs Composición
3. Código Sucio por Clases Alternativas con Interfaces Diferentes
4. Código Sucio por Herencia Rechazada
5. Código Sucio por Jerarquías Paralelas de Herencia
6. Patrón Método Plantilla
7. Técnica del Doble Despacho
8. Principio de Separación de Interfaces

5.5.1. Principio de Sustitución de Liskov

- Definido por **Robert Martin** (*Liskov's Sustitution Principle - **LSP***) como uno de los principios SOLID
  - Está inspirado en los trabajos de **Barbara Liskov**: *"Lo que se quiere aquí es algo como la siguiente propiedad de sustitución: si para cada objeto o_1 de un tipo S, hay un objeto o_2 de tipo T tal que para todo progama P definido en términos de T, el comportamiento de P no cambia cuando o_1 es sustituido por o_2, entonces S es un subtipo de T"*
  - Se cumple sólo cuando los tipos de derivados son totalmente sustituibles por sus tipos base de forma que las funciones que utilizan estos tipos base pueden ser reutilizados con impunidad y los tipos derivados se puede cambiar con impunidad.
  - ***El Principio de Sustitución de Liskov dice que las funciones que usan punteros o referencias a una clase base debe ser capaz de usar los objetos de las clases derivadas sin conocerlas***.

5.5.1. Principio de Sustitución de Liskov

- Por tanto, la relación de herencia se refiere al comportamiento. No al comportamiento privado intrínseco sino al comportamiento público extrínseco del que dependen los clientes
  - ***El Principio de Sustitución de Liskov dice que se cumple cuando se redefine un método en una derivada reemplazando su precondición por una más débil y su postcondicion por una más fuerte***
    - La precondición de un subtipo es creada combinando con el operador OR las precondiciones del tipo base y del subtipo, lo que resulta una precondición menos restrictiva.
    - La postcondición de un subtipo es creada combinando con el operador AND las postcondiciones del tipo base y del subtipo, lo que resulta una postcondición más restrictiva.

5.5.1. Principio de Sustitución de Liskov

- Violaciones:
  - Una de las violaciones más evidentes de este principio es el uso de la Información de Tipos en Tiempo de Ejecución (*instantceof, RTTI, …*) para seleccionar una función basada en el tipo de un objeto. Muchos ven esta estructura como el anatema de la Programacion Orientada a Objetos.
  - Cuando se considera si un diseño particular es apropiado o no, no se debe simplemente ver la solución aislada. Uno debe verlo en términos de las asunciones razonables que serán hechas por los usuarios de este diseño. Por ejemplo:
    - Si Square hereda de Rectangle redefiniendo los métodos para cambiar el ancho y alto cambiando el otro para mantener la invariante del Square, se incumple la asunción de los clientes con su clase padre que no esperan que un cambio del ancho repercuta bajo ningún concepto en el alto.

5.5.2. Relación de Herencia vs Composición

- Contexto:
  - La Relación de Composición responde a **A tiene un B**
  - La Relación de Herencia responde a **A es un B**
- La possible elección viene dada porque:
  - **Mientras que tener no es siempre ser**. Por ejemplo: un propietario de un coche es una persona pero no es un coche; un propietario de un coche tiene un coche
  - **En muchos casos ser también es tener**. Por ejemplo: un ingeniero del software es un ingeniero, o sea que en cada ingeniero del software hay un ingeniero, o sea, un ingeniero del software tiene un ingeniero
  - Siempre que se analiza/diseña una relación de herencia se puede analizar/diseñar su contrapartida como relación de composición

5.5.2. Relación de Herencia vs Composición

- Justificación:
  - **Incumplimiento del Principio de Sustitución de Liskov**
  - **Regla de Cambio**. No usar herencia para describir un relación percibida como ISA si el correspondiente objeto compuesto puede cambiar en tiempo de ejecución. La relación de composición permite el cambio y la herencia no.
  - **Reglas de Polimorfismo.** La herencia es apropiada para describir un relación percibidad ISA si una entidad de un tipo más general puede llegar a ser enlazado a objetos de un tipo más especializado

5.5.2. Relación de Herencia vs Composición

- Solución:
  - **Delegación.** Dos objetos están involucrado en manejar una petición: un objeto la recibe y delega la operación a su delegado. La clase padre juega el rol de parte delegada y la clase hija juega el rol del todo donde:
    - Los métodos transmitidos de la clase padre a la hija se delegan del todo a la parte delegada con la misma cabecera de método
    - Los métodos redefinidos de la clase hija reutilizando o no el método de la clase padre se codifican en la clase todo y delegan o no según la necesidad de reutilización
    - Los métodos abstractos de la clase padre no se implantan en la parte delegada
  - **Interfaces.** Si al desechar la herencia se imposibilita un polimorfismo necesario entre las clases implicadas en la jerarquía original, se puede heredar de interfaces que habilitan igualmente el polimorfismo.

5.5.3. Código Sucio por Clases Alternativas con Interfaces Diferentes

- Problema:
  - Complejidad innecesaria aumentando el espacio de nombres a manejar
  - Impide el polimorfismo
- Solución:
  - Cambiar el nombre de los métodos implicados con cabeceras diferentes pero que hacen lo mismo semánticamente, aunque programáticamente sea diferente.
  - A menudo, esto no es suficientemente. Puede ser posible mover responsabilidades para conseguir que varios métodos hagan lo mismo semánticamente.
  - Si tiene cabida, extraiga el código común en una clase padre

5.5.4. Código Sucio por Herencia Rechazada

- Justificación:
  - Las subclases heredan los métodos y atributos de sus padres que no necesitan.
- Solución:
  - La solución gira en torno a crear clases intermedias en la jerarquía, habitualmente abstractas, mover métodos y atributos hacia arriba y hacia abajo hasta que todas las subclases reciban los métodos y atributos necesarios y no más. De tal manera que cada clase padre tenga el factor común de sus clases derivadas.
  - A menudo, esta solución complica la jerarquía en exceso. En tal caso, si la herencia rechazada es la implantación "vacía" de un método de la clase derivada podría considerarse como solución frente a la complicación de la jerarquía. Pero si la herencia rechazada es la transmisión de métodos públicos implantados a clases derivadas que no lo necesitan, debería re-diseñarse la jerarquía de herencia por composición por delegación para evitar corromper la interfaz de la clase derivada

5.5.4. Código Sucio por Herencia Rechazada

- Justificación:
  - Las subclases heredan los métodos y atributos de sus padres que no necesitan.
- Solución:
  - La historia tradicional es que esto significa que la jerarquía está mal. Es necesario crear una nueva clase de hermanos y utilizar Empuje hacia abajo Método y empuje hacia abajo Campo de empujar todos los métodos utilizados para el hermano. De esa manera el padre tiene sólo lo que es común. A menudo se escucha consejos que todas las superclases deben ser abstracto.
  - Usted adivinas de nuestro uso sarcástico de lo tradicional que nosotros no vamos a asesorar a esto, al menos no todo el tiempo. Nosotros subclases volver a utilizar un poco de comportamiento todo el tiempo, y nos encontramos con un perfectamente buena forma de hacer negocios. Hay un olor, no podemos negarlo, pero por lo general no es un olor fuerte. Por eso decimos que si el legado negado está causando confusión y problemas, siga el consejo tradicional.
  - Sin embargo, no se siente que tiene que hacer todo el tiempo. Nueve de cada diez veces el olor es demasiado débil para ser digno de limpieza.
  - El olor del legado negado es mucho más fuerte si la subclase está reutilizando comportamiento, pero no quiere apoyar a la interfaz de la superclase. No nos importa negarse implementaciones, pero negándose interfaz nos mete en nuestros altos caballos. En este caso, sin embargo, no jugar con la jerarquía; quiere destripar que aplicando Reemplazar Herencia con Delegación.

5.5.5. Código Sucio por Jerarquías Paralelas de Herencia

- Justificación:
  - Es un caso especial de la Cirugía a Escopetazos.
  - Cada vez que haces una subclase de una clase, también tienes que hacer una subclase de otra. Se reconoce por los prefijos en los nombres de clases de la jerarquía son los mismos que los prefijos de la otra jerarquía
    - Clase Paciente
      - Clase Paciente de Seguridad Pública
        - Clase Paciente de Seguridad Pública de Traumatología
        - Clase Paciente de Seguridad Pública de Cardiología
        - …
      - Clase Paciente de Seguridad Privada
        - Clase Paciente de Seguridad Privada de Traumatología
        - Clase Paciente de Seguridad Privada de Cardiología
        - …
      - …

5.5.5. Código Sucio por Jerarquías Paralelas de Herencia

- Solución:
  - Reestructurar la jerarquía:
    - Reubicar responsabilidades
    - Aplicar el Patrón Método Plantilla
    - Una clase contiene tantos roles polimórficos como cada una de las jerarquía paralelas en los que delega y combina su comportamiento
      - Clase *Paciente* tiene un Enfermo y un Seguro
        - Clase *Seguro*
          - Clase *Seguro Público*
          - Clase *Seguro Privado*
          - …
        - Clase *Enfermo*
          - Clase *Enfermo de Traumatología*
          - Clase *Enfermo de Cardiología*
          - …

5.5.6. Patrón Método Plantilla

- Sinónimo
  - ***Template Method*** [Patrón de Diseño; **Gamma** et al]
- Justificación:
  - Se dificulta la extracción de un factor común en los códigos de los métodos de las clases derivadas por detalles inmersos en el propio código pero respetando un esquema general
- Solución:
  - Definir el esqueleto de un algoritmo de un método, diferir algunos pasos para las clases derivadas. El patron permite que las clases derivadas redefinan esos pasos abstractos sin cambiar la estructura del algoritmo de la clase padre

5.5.7. Técnica de Doble Despacho

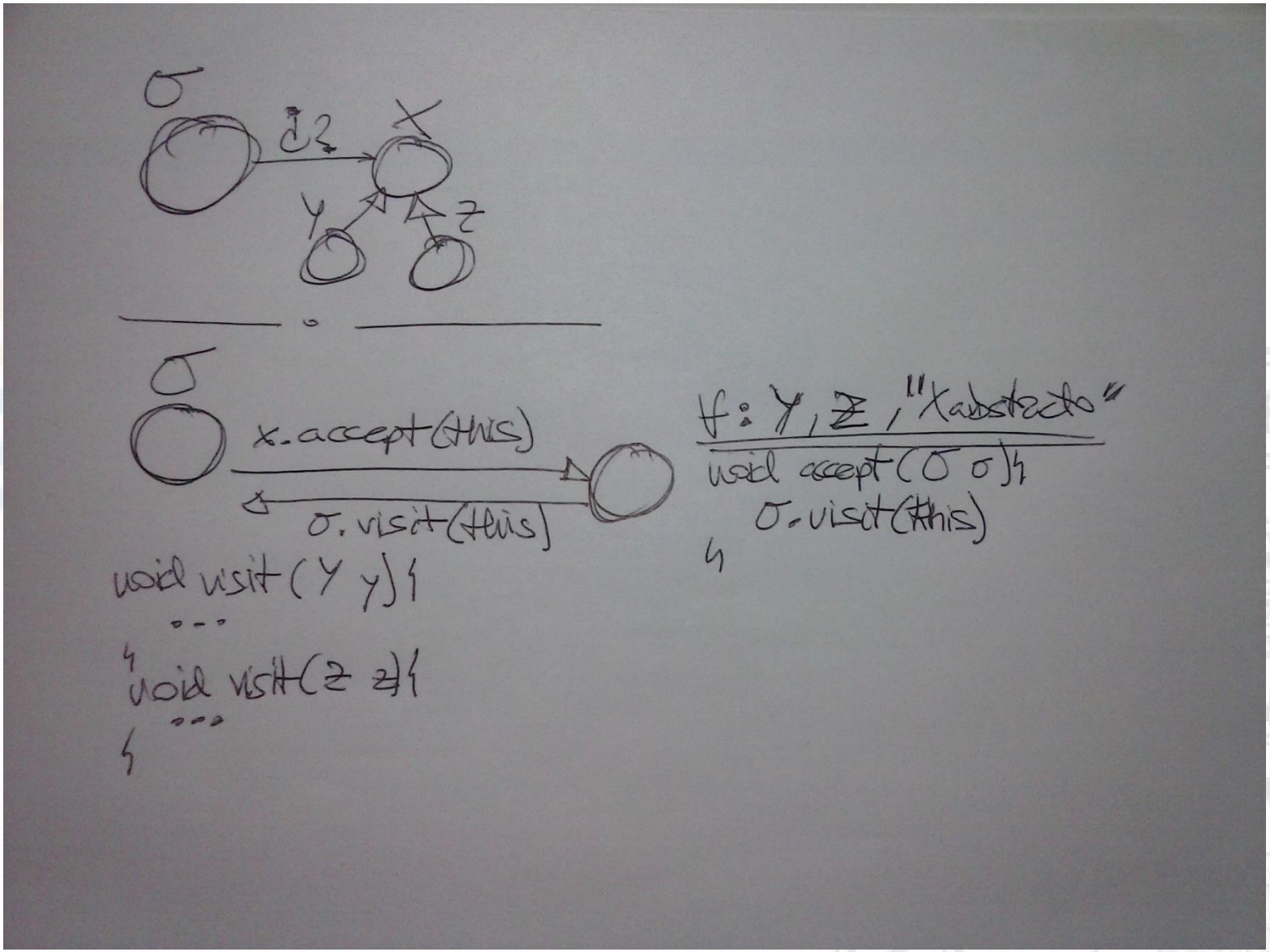
- Motivación:
  - Necesidad de dar un tratamiento específico según la clase derivada concreta de un objeto polimórifco
- Solución:
  - Enviar un mensaje *accept* al objeto polimórfico auto-pasándose como parámetro (*this*)
  - Codificar en las clases de la jerarquía del objeto polimórifco el método anterior con la siguiente implantación:

```
void accept(X x){
    x.visit(this);
}
```

  - Codificar en la clase que desea realizar el tratamiento por cada clase derivada un método por cada una:

```
void visit(Derivada derivada){
    …
}
```

5.5.7. Técnica de Doble Despacho



5.5.8. Principio de Separación de Interfaces

- Definido por **Robert Martin** (*Interface Segregation Principle - **ISP***) como uno de los principios SOLID
- Motivación:
  - Cuando un cliente depende de una clase que contiene una interfaz que no usa pero otros clientes sí la usan, el primer cliente será afectado por cambios que otros clientes fuercen sobre la clase que da el servicio.
  - En una jerarquía de herencia a veces se fuerza a incorporar interfaces únicamente por el beneficio de una de sus subclases. Esta práctica es indeseable porque cada vez que una clase derivada necesite una nueva interfaz, ésta será añadida a la clase base. Esto va a contaminar aún más la interfaz de la clase base, por lo que es "gorda".
  - Además, cada vez que un nuevo interfaz se añade a la clase base, éste debe ser implementado (o permitido por defecto) en las clases derivadas. De hecho, una práctica asociada es añadir estos interfaces a la clase base con métodos "vacíos" más que con métodos abstractos, así las clases derivadas no son agobiadas con su necesaria implementación; lo cual viola el principio de sustitución de Liskov
  - ***El Principio de Segregación de Interfaces dice que Los clientes no deberían forzarse a depender de interfaces que no usan***

5.5.8. Principio de Separación de Interfaces

- Solución:
  - Sería deseable evitar el acoplamiento entre clientes como sea possible y separar interfaces como sea possible. Dado que los clientes están "separados", las interfaces deben permanecer también "separados".
  - Las clases que tienen interfaces "gordas" son clases cuyos interfaces no son cohesivos. En otras palabras, el interfaz de una clase puede ser rota en grupos de funciones. Cada grupo sirve a diferentes conjuntos de clientes. Así, algunos clientes usan un grupo de funciones y otros clientes usan otro grupo.
  - El ISP reconoce que hay objetos que requieren interfaces no cohesivos, sin embargo sugiere que los clientes no deberían conocerlos como una única clase. En cambio, los clientes deberían conocer clases base abstractas que tengan interfaces cohesivas.

5.5.8. Principio de Separación de Interfaces

- Solución:
  - Estas interfaces deben ser implementadas en el mismo objeto dado que la implementación de ambos interfaces manipulan los mismos datos. La respuesta a esto radica en el hecho de que los clientes de un objeto no necesitan acceder a ella a través de la interfaz del objeto. Más bien, se puede acceder a él a través de la delegación, o por medio de una clase base del objeto.
    - Por ejemplo, una secretaría de universidad ofrece multitud de servicios variopinto a distintas entidades: alumnos, profesores, dirección, rectorado, … Es el mismo objeto trabajando sobre los mismos atributos pero puede implementar diversos interfaces enfocados a cada entidad: secretaría de alumnos ofrece matricularse, expediente académico, ..; secretaría de profesores ofrece cerrar un grupo, firmar actas, …; secretaría de dirección ofrece configurar planes de estudios, … Así, los distintos clientes colaboran con el mismo objeto pero con interfaces completamente diferentes

5.5.8. Principio de Separación de Interfaces

- Compromiso:
  - Como todos los principios SOLID se requiere una gasto de esfuerzo y tiempo adicional para aplicar durante el diseño e incrmenta la complejidad del código. Pero produce un diseño flexible. Si lo aplicamos más de lo necesario resultará un codigo llene de interfaces con un solo método. Asi que su aplicacion sería hecha en base a nuestra experiencia y sentido común identificando área donde la extension de código es más posible en el futuro: YAGNI!

6. Diseño de Dependencias

1. Código Sucio por Inapropiada Intimidad
2. Leyes de Demeter
3. Inversión de Control
4. Inyección de Dependencias
5. Principio de Inversión de Dependencias
6. Principio Abierto/Cerrado

5.6.1. 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 intereses en común, extrae en una nueva clase poniendo lo 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 privada de otras clases. Nosotros pensamos que nuestras clases deberían ser estrictas con reglas puritanas" **[Fowler]**

5.6.2. Leyes de Demeter

- Sinónimos:
  - Principio "No hables con extraños" [**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.

5.6.3. Inversión de Control

- Sinónimo:
  - Principio Hollywood: "No me llames, ya te llamaremos"
- Justificación:
  - *"Una característica importante de un framework es que los métodos definidos por el usuario para adaptar el framework, a menudo se llaman desde dentro del propio framework , en lugar desde el código de la aplicación del usuario. El framework a menudo desempeña el papel del programa principal en la coordinación y secuenciación de actividad de la aplicación. Esta **inversión de control** da al framework el poder para servir como esqueletos extensibles. Los métodos suministrados por el usuario se adaptan a los algoritmos genéricos definidos en el framework para una aplicación particular"* [**Johonson & Foote** 1988]

5.6.3. Inversión de Control

- La **Inversión de Control** es una parte fundamental de lo que hace un *framework* diferente a una biblioteca.
  - Una biblioteca es esencialmente un conjunto de funciones que se pueden llamar, en estos días por lo general organizados en clases. Cada llamada que hace un poco de trabajo y devuelve el control al cliente.
  - Un *framework* encarna algún diseño abstracto, con un comportamiento más integrado. Para utilizarlo es necesario insertar comportamiento en varios lugares en el *framework* ya sea por subclases o por conectar sus propias clases. El código del *framework* después llama a ese código en estos puntos.

5.6.3. Inversión de Control

- Variaciones:
  - **Patrón Método Plantilla** con redefinición de métodos abstractos
  - **Inyección de Dependencias**: *"algunas personas confunden el principio general de Inversión de Control con los estilos específicos de Inversión de Control, como la Inyección de Dependencias, que estos contenedores utilizan"* [**Fowler**]
  - **Eventos** con auditores que determinan su comportamiento
  - **Configuración** con datos externos al *framework* para determinar el comportamiento

5.6.4. Inyección de Dependencias

- Sinónimo:
  - Pluggin
- Justificación:
  - Eliminar las dependencias de una clase de aplicación hacia la implementación de otra clase, servicio, para que esta clase sea reutilizada por implementaciones alternativas del servicio actual.
    - Por ejemplo: una clase que filtra ciertas entidades a partir de un listado de dichas entidades obtenido desde un fichero de texto no quiere acoplarse a ese listado y poder reutilizarse con un listado obtenido desde XML, una base de datos, un servicio remoto, …
  - Lo primero será que la clase trabaje con un interfaz del servicio para que éste pueda ser extendido por otras implementaciones de servicio diferentes. Pero, aunque la clase guarde la referencia al objeto servicio a través de un interfaz, mientras la clase instancie directamente el objeto servicio, continuará el acoplamiento indeseado que impide la reutilización.
    - El problema persiste mientras la clase que filtra instancie un objeto concreto para obtener el listado

5.6.4. Inyección de Dependencias

- Solución:
  - Lo segundo será inyectar (hacer accesible) el servicio concreto a la clase a través de la interfaz de tal manera que por la abstracción del polimorfismo desconoce con qué servicio concreto está trabajando. Alternativas:
    - Por el constructor, muy recomendable, porque conviene, tanto sea posible, crear objetos válidos en el momento de la construcción
    - Por métodos *setter*, cuando por constructor se complica por la cantidad de parámetros, muchas formas de construcción, cuando se desea cambiar dinámicamente el proveedor del servicio durante la vida del objeto al que se le inyectó
  - Por último, para la creación e inyección del servicio a la clase:
    - Para aplicaciones que pueden desplegarse en muchos lugares, un archivo de configuración XML tiene más sentido.
    - Para aplicaciones sencillas que no tienen demasiada variación en el despliegue, es más fácil usar código para el ensamblado de los componentes.

5.6.5. Principio de Inversión de Dependencias

- Definido por **Robert Martin** (*Dependency Inversion Principle - **DIP***) como uno de los principios SOLID
  - Se puede entender como el resultado de aplicar rigurosamente los principios de Sustitución de Liskov y Abierto/Cerrado.
  - ***Los módulos de alto nivel no deberían depender de los módulos de bajo nivel. Ambos deberían depender de abstracciones***
    - En vez de que un *Copier* (modulo de alto nivel) lea de un *KeyboardReader* y escribe en *PrintWriter* (módulos de bajo nivel), debería leer de una interfaz *Reader*, base de *KeyboardReader* y escibir en una interfaz *Writer*, base de *PrintWriter* de tal forma que *Copier*, *KeyboardReader* y PrintWriter *dependan* de las abstracciones *Reader* y *Writer*
  - Cuando los módulos de alto nivel son independientes de los de bajo nivel, se pueden reutilizar los primeros con sencillez. En este caso, la instanciación de las objetos de bajo nivel dentro de la clase de alto nivel no puede ser hecha con el operador *new*
    - La lógica de *Copier* será reutilizada sin cambios cuando nuevos dispositivos de entrada y salida entren en juego con el cambio de requisitos heredando de las interfaces *Reader* y *Writer*.

5.6.5. Principio de Inversión de Dependencias

- ***Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones***
  - Las clases abstractas no deberían depender de las clases concretas. Las clases concretas deberían depender de las clases abstractas
  - En vez de que un *LampButton* dependa de una *Lamp* para encenderla y apagarla, un *ToggleButton* depende de un *ToggleClient* que puede encenderse y apagarse podrá reutilizarse por cualquier botón, *LampButton*, … que herede de *ToggleButton* para encender y apagar cualquier cliente, *Lamp*, … que herede de *ToggleClient*.
- Cuando las abstracciones son independientes de los detalles, se pueden reutilizar los primeros con sencillez. En este caso, las abstracciones no pueden mencionar ninguna clase derivada
  - Las clases derivadas únicamente redefinirán cómo se aprieta y libera el botón particular y cómo se enciende y apaga el cliente particular reutilizando toda la lógica de las abstracciones *ToggleButton y ToggleClient*.

5.6.5. Principio de Inversión de Dependencias

- Usar este principio implica un incremento de esfuerzo, porque resultarán más clases e interfaces para mantener, código más complejo pero más flexible. Este principio no sería applicable a ciegas en cada clase o cada módulo.
  - Si se tiene la funcionalidad de una clase que es más que possible que no cambie en el futuro, no hay necesidad de aplicar este principio: YAGNI!

5.6.6. Principio Abierto/Cerrado

- Definido por **Robert Martin** (*Open/Close Principle* - ***OCP***) como uno de los principios SOLID
  - El autor original es **Bertran Meyer** en 1988
- Motivación:
  - Se debería diseñar módulos que nunca cambien. Cuando los requisitos cambian, se extiende el comportamiento de dichos módulos añadiendo nuevo código, no cambiando el viejo código que ya funciona
- Justificación:
  - **Las entidades de software (módulos, clases, métodos, …) deberían estar abiertas a la extension pero cerradas a la modificación**
  - Parece que estos dos atributos están en conflicto entre sí. La forma normal de extender el comportamiento de un módulo es hacer cambios a ese módulo. Un módulo que no puede ser cambiado se piensa normalmente que tendrá un comportamiento fijo.

5.6.6. Principio Abierto/Cerrado

- Solución:
  - Usando los principios de la programación orientada a objetos, es possible crear abstracciones que son fijas y a la vez representan un grupo ilimitado de posibles comportamientos.
    - Las abstracciones son clases base abstractas y el ilimitado grupo de posibles comportamientos es representado por todos las posibles clases derivadas. Es possible para un modulo manipular una abstracción. Tal modulo puede ser cerrado para la modificación si depende de una abstracción que es fija. Todavía el comportamiento del modulo puede ser extendido creando nuevas derivadas de la abstracción.
  - No preguntar por el tipo de objeto polimórfico
  - No usar atributos que no sean privados
  - No usar variables globales

5.6.6. Principio Abierto/Cerrado

- Contraindicaciones:
  - Debería estar claro que no significa que un programa sea 100% cerrado. En general, no es la cuestión cómo cerrar un modulo, habrá siempre alguna clase de cambio para la cual no está cerrado
    - Dado que el cierre no puede ser complete, debe ser una estrategia. Estos es, los diseñadores deben elegir la clase de cambios contra los cuales cerrar el diseño. Esto toma cierta cantidad de presciencia derivada de la experiencia. Los diseñadores experimentados conocen a los usuarios y la industria suficientemente bien para juzgar la probabilidad de diferentes clases de cambios. Se asegura de que el Principio Abierto/Cerrado es aplicado para los cambios más probables: YAGNI!

7. Resumen de Métricas

- Tamaño:
  - Clases:
    - **Líneas de código** [0 - ∞): **[250 – 500] máximo**
    - **Número de atributos** [0 - ∞): **[3 – 5] máximo**
    - **Número de métodos** [0 - ∞): **[20 – 25] máximo**
  - Métodos
    - **Número de parámetros** [0 - ∞): **[3 – 5] máximo**
    - **Complejidad Ciclomática** [0 - ∞): **[8 – 16] máximo**
    - **Estructuras anidadas** [0 - ∞): **[2 – 3] máximo**
    - **Número de lineas de código** [0 - ∞): **[10 – 15] máximo**
- Cohesión:
  - **Falta de Cohesión (LCOM)** [0 - 1]: **~ 0**
- Acoplamiento:
  - **Acomplamiento eferente (CBO)** [0 - ∞): **[3-5] máximo**