Tecnología de Programación:

Resumen de teoría

Principios SOLID

The Single Responsability Principle

- Nunca debe haber más de una razón para que una clase cambie.

¿Qué es una responsabilidad?

En el contexto del SRP definimos una responsabilidad como una “razón de cambio”. Si pensamos que hay más de un motivo para que una clase cambie entonces esa clase tiene más de una responsabilidad.

¿Por qué es importante separar las responsabilidades en clases distintas?

Porque cada responsabilidad es un eje de cambio. Cuando los requerimientos cambian, ese cambio se va a manifestar a través de un cambio en las responsabilidades entre las clases. Si una clase asume más de una responsabilidad entonces va a tener más de una razón para cambiar además que si esto sucede, las clases se acoplan.

The Open-Closed Principle

Las entidades de software (clases, módulos, funciones, etc) deberían ser abiertas para su extensión pero cerradas para modificación. Cuando los requerimientos cambian, se extiende el comportamiento de los módulos añadiendo nuevo código, no cambiando el código viejo que ya funcionaba.

- Es **“abierto para extensión”**: Esto significa que el comportamiento del módulo puede ser extendido. Esto puede hacer que el módulo pueda comportarse de nuevas y diferentes maneras a medida que los requerimientos de la aplicación cambian.
- Es **“cerrado para la modificación”**: El código fuente del módulo es inviolable. Nadie tiene permitido hacer cambios en este.

Hacer todas las variables privadas: Las variables de una clase no deberían ser conocidas por ninguna otra clase incluyendo clases derivadas. Estas deberían ser declaradas *private* em vez de *public* o *protected.* Cuando una variable de una clase cambia, cada función que depende de esa variable debe cambiar. De esta manera ninguna función que dependa de una variable puede ser cerrada respecto a la misma.

Nunca usar variables globales: Ningún módulo que depende de una variable global puede ser cerrado en contra de otro módulo que quizás puede escribir esa variable. Cualquier módulo que utilice la variable de una manera no esperada por los otros módulos puede provocar que estos fallen. En estos casos el **Open-Closed Principle** no es violado pero esto significa una clara violación de estilo.

Conclusión: El **Open-Closed Principle** es el corazón del diseño orientado a objetos. Confirme a este principio podemos decir que cede los principales beneficios de la tecnología orientada a objetos: *reusabilidad* y *mantenibilidad.* Este requiere dedicación por parte del diseñador para aplicar la abstracción necesaria a esas partes del programa que el diseñador cree que están sujetas a cambios.

The Liskov Substitution Principle

- Funciones que usan punteros o referencias a clases bases deben ser capaces de usar objetos de clases derivadas sin conocerlos.

Si existe una función que usa un puntero o una referencia a una clase base (que no conforma el **Liskov Substitution Principle**) entonces esta debe conocer todas los derivados de una clase base. Ese tipo de funciones violan el **Open-Closed Principle** porque esta se debe modificar cada vez que un nuevo derivado de la clase base es creado.

Existe una fuerte relación entre el **Liskov Substitution Principle** y **Diseño por contrato** de Bertrand Meyer. Usando este diseño decimos que cada método de una clase declara precondiciones y postcondiciones. Las precondiciones deben ser verdaderas cuando el método se ejecuta y las postcondiciones son garantizadas por el método al finalizar su ejecución.

Cuando se redefine una rutina solo hay que reemplazar su precondición por una más débil y su postcondición por una más fuerte.

Conclusión: El principio de substitución de Liskov es una importante característica de todos los programas que cumplen con el **Open-Closed Principle.** Solo cuando los tipos derivados son completamente substituibles por sus tipos bases y las funciones pueden usar esos tipos bases con impunidad y los tipos derivados pueden cambiar con impunidad.

The Dependency Inversion Principle

¿Cuándo un software presenta un mal diseño?

Si un software presenta alguno de estos tres rasgos entonces presenta una mal diseño:

1) Es difícil de modificar porque cada cambio afecta a muchas partes del sistema (Rigidez)
2) Cuando se realiza un cambio, se rompen partes inesperadas en el sistema (Fragilidad)
3) Es muy difícil de reusar en otras aplicaciones ya que no puede ser desenredado de la aplicación actual (Inmovilidad)

Causas del mal diseño.

Un diseño es rígido si no es posible cambiarlo fácilmente. Ese tipo de Rigidez se debe a que un simple cambio en un software fuertemente interdependiente genera una cascada de cambios en los módulos dependientes. El impacto del cambio es imposible de estimar lo que produce que el costo que este requiere sea imposible de predecir.

La fragilidad es la tendencia de un programa de romperse en muchos lugares cuando se realiza un solo cambio. Los nuevos problemas pueden surgir incluso en áreas del sistema que no tienen ninguna relación conceptual con el área que cambió.

Un diseño es inmóvil cuando las partes deseadas de un sistema son muy dependientes de pares no deseadas del mismo. En muchos casos dichos diseños no son reusados ya que se estima que el costo de separación será mucho más alto que el costo de re-desarrollar el diseño desde cero.

A) Módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deberán depender de abstracciones.
B) Las abstracciones no deberían depender de los detalles. Los detalles deben depender de las abstracciones.

Consideremos las implicaciones de que los módulos de alto nivel dependan de módulos de bajo nivel. Los módulos de alto nivel contienen las políticas y los modelos del negocio de la aplicación. Estos módulos contienen la identidad de la aplicación. Ahora, si estos dependen de módulos de bajo nivel, un cambio en estos últimos tendrá un impacto directo en los módulos de alto nivel, forzándolos a cambiar. Esto es absurdo. Los módulos de alto nivel deberían forzar a los módulos de bajo nivel a cambiar. Además, son los módulos de alto nivel los que buscamos que sean capaces de ser reusables.

- Este principio es el corazón de los frameworks.
- Desde que se comienza a aislar las abstracciones de los detalles, el código se vuelve mucho más fácil de mantener.

The Interface Segregation Principle

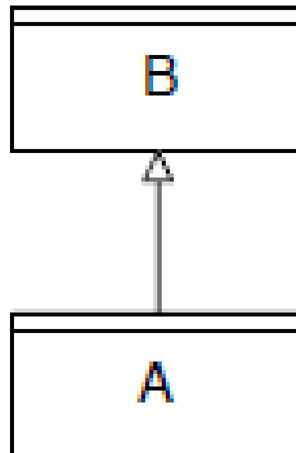
Los clientes no deberían ser forzados a depender de interfaces que no van a usar. Cuando esto sucede, los clientes están sujetos a los cambios de esas interfaces. Esto produce un acoplamiento involuntario entre los clientes.

Cuando un cliente depende de una clase que contiene interfaces que ese cliente no usa pero que otro cliente si usa entonces ese cliente va a estar afectado a los cambios que otros vayan a realizar sobre la clase.

Conclusión: Las *“interfaces gordas”*, es decir, las interfaces que no son específicas para un único cliente, tienen muchas desventajas. Se produce un acoplamiento no deseado entre clientes que deberían estar aislados. Usando el patrón ***ADAPTER*** o a través de delegación (objetos) o herencia múltiple (clases)las interfaces gordas pueden ser separadas en clase abstractas que rompen el acoplamiento no buscado entre clientes.

Subtipos y subclases

A partir del siguiente diagrama:



Podemos decir que A y B son módulos con objetos y métodos asociados. Decimos que A es un B si todo objeto A es un objeto B.

Esta relación de subconjunto es una condición necesaria pero no suficiente para afirmar que es una relación de subtipo. Un tipo A es subtipo de un tipo B cuando la especificación de A implica la especificación de B. Es decir, cualquier objeto (o clase) que satisface la especificación de A entonces también satisface la especificación de B porque esta última es más débil.

La definición de subtipo depende de la definición de especificación que manejemos (fuerte o débil). Se define como subtipo verdadero si en cualquier parte del código si se espera un objeto B, un objeto A es aceptado. El código escrito para funcionar con objetos de tipo B garantiza que va a seguir funcionando si se reemplaza por objetos de tipo A. Además, el comportamiento será el mismo si consideramos que los aspectos del comportamiento de A están incluidos en el comportamiento de B.

Principio de sustitución

Este principio provee una definición precisa de cuando dos tipos son subtipos. Informalmente dice que los subtipos deben ser sustituibles por supertipos. Esto garantiza que si el código depende de un supertipo, pero un subtipo es sustituido, el comportamiento del sistema no se debería ver afectado.

Para los métodos existen dos propiedades:

1) Por cada método en el supertipo el subtipo debe tener el correspondiente método
2) Cada método en el subtipo que corresponde a uno del supertipo:
   a. Requiere menos (tiene una precondición más débil)
      i. No requiere más clausulas y cada una de ellas no es más estricta que una en el método del supertipo.
      ii. El tipo de los argumentos pueden ser supertipos de los que hay en el supertipo. Esto se llama contravarianza. Por ejemplo: si se espera un argumento de tipo Bicicleta, puede recibir un argumento de tipo Vehículo.
   b. Garantiza más (tiene una postcondición más fuerte)
      i. No hay más excepciones
      ii. No hay más variables modificadas
      iii. En la descripción del resultado hay más cláusulas y estas describen propiedades más fuertes.
      iv. El tipo del resultado puede ser un subtipo del supertipo. Esto se llama covarianza: el tipo retornado por el método del subtipo es un subtipo del tipo al ser retornado en el método del supertipo.

Propiedades:

- Cualquier propiedad que es garantizada por el supertipo también debe ser garantizada por el subtipo. (el subtipo permite una propiedad más fuerte)

Java Subtypes

Los tipos en Java son clases, interfaces y primitivas. Java tiene su propia noción de subtipo (que solo incluye clases e interfaces). Esta es una noción más débil que la del verdadero subtipo descrito anteriormente. Los subtipos en Java no necesariamente satisfacen el principio de sustitución. Incluso puede suceder que un subtipo que satisface el principio no sea permitido en Java.

Para que un tipo sea subtipo de otro en Java la relación debe estar declarada (implements o extends) y los métodos deben cumplir dos propiedades similares a la de los subtipos verdaderos pero más débiles:

1) Por cada método en el supertipo, el subtipo también debe tenerlo.
2) Para cada método en el subtipo que corresponde con uno del supertipo:
    a. Los argumentos deben ser del mismo tipo
    b. El resultado debe ser del mismo tipo
    c. No hay más excepciones declaradas.

Java subclassing

Las subclases tienen grandes ventajas, todas ellas son el pilar de la reusabilidad:

- Las implementaciones de las subclases no necesitan repetir campos ni métodos que no cambian. Pueden usarlos de la superclase.
- Los clientes no necesitan modificar el código cuando un nuevo subtipo es añadido pero puede reusar el código existente.
- El diseño resultante termina teniendo más modularidad y se reduce su complejidad, ya que tanto los diseñadores como los implementadores solo tienen que entender el supertipo y no cada subtipo.

Diseño por contrato

El documento define confiabilidad como una combinación entre correctitud y robustez, más precisamente, la ausencia de bugs.

- La posta de la tecnología orientada a objetos es la reusabilidad.
- La confiabilidad es un componente centrar el cualquier definición de calidad de software.
- Un conjunto coherente de principios metodológicos ayudan a producir software correcto y robusto.
- Un mejor entendimiento de la herencia y sus técnicas asociadas (redefinición, polimorfismo y ligadura dinámica)

Programación defensiva: Aconseja que las rutinas sean lo más generales posibles ya que las rutinas parciales son consideradas peligrosas porque podrían producir consecuencias inesperadas si la persona que la llamó no tolera las reglas. Esta técnica muchas veces viola su propio propósito ya que, añadiendo código redundante solo “por si acaso” contribuye a la complejidad del software.

Para aplicar el diseño por contrato en la construcción de un software existe un mecanismo que expresa condiciones para especificar la relación entre cliente y el proveedor. Este mecanismo se llama aserción. Algunas aserciones son llamadas precondiciones y otras postcondiciones y se aplican a rutinas individuales, otras, llamadas invariantes de clase, obligan a todas las rutinas a cumplir una condición.

Cuando se produce una excepción se pueden producir tres respuestas por parte del sistema:

1) Si la estrategia falló quizás pueda haber otra disponible. “Perdimos una batalla pero no la guerra”. En este caso, la rutina debería poner los objetos en un estado consistente y realizar otro intento, usando una nueva estrategia. Esto se llama *Reanudación* (Resumption)
2) Sin embargo, puede que también hayamos perdido la guerra. No hay ninguna nueva estrategia disponible. Cuando esto sucede, la rutina debe poner los objetos en un estado consistente, rescindir el contrato y reportar la falla al cliente. Este mecanismo se llama *pánico organizado*.

3) La última respuesta es más la menos probable que suceda: *"La falsa alarma".* Esto debería ocurrir solo para señales del sistema operativo o del hardware. (ver ejemplo del sistema multiventana)

Observación: Cualquier excepción capturada, ya sea por reanudación o por pánico organizado debe restaurar la invariante.

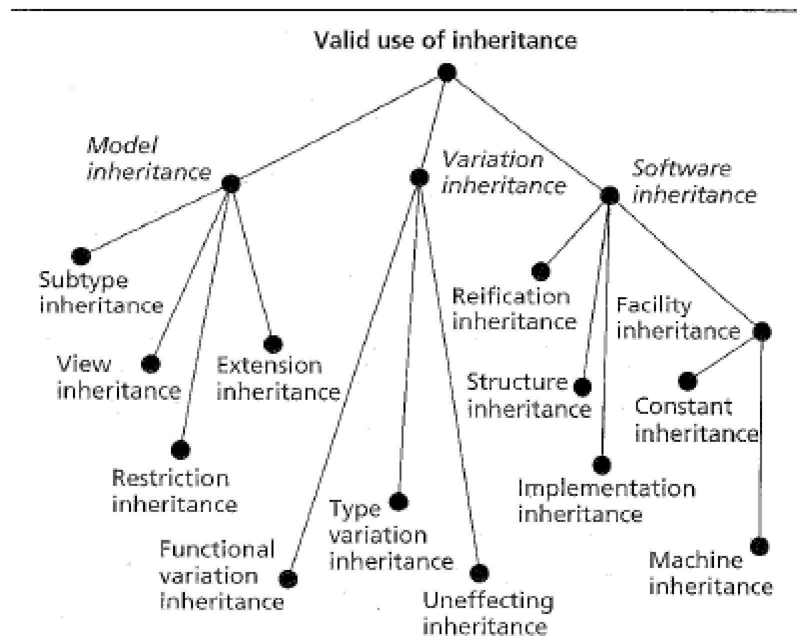
Las diferentes caras de la herencia

Usos **inapropiados** de la herencia:

- Hay una relación que no es la relación "es un". Ejemplo: Arbol_manzanero que hereda de Árbol y Manzana.}
- **Taxonomía:** es una práctica común de los principiantes. Estos agregan nodos intermedios que son inútiles solo para representar una condición booleana. La teoría de tipos abstractos provee la forma correcta de usar la herencia: Solo introducir una nueva clase si esta provee más funcionalidades, ya sean nuevas o modificadas.
- **Herencia por conveniencia:** El desarrollador ve características útiles en una clase entonces hereda de ella para reusar esas características pero sin la relación "es un" entre las correspondientes abstracciones.

Usos **correctos** de la herencia:

- **Herencia de modelo**, la cual refleja la relación "es un " entre las abstracciones en el modelo.
- **Herencia de software**, la cual expresa la relación entre el software con si mismo más que con el modelo.
- **Herencia de variación**, la cual describe como una clase difiere de otra.



-

Herencia de subtipo: A y B representan los conjuntos A' y B' que son objetos externos. A es diferido, B' es un subconjunto de A' y el conjunto modelado por cualquier subtipo heredero de A es disjunto de B'

Herencia de Visión: B describe la misma abstracción que A pero vista desde un ángulo diferente.

Herencia de restricción: Las instancias de B son aquellas instancias de A que satisfacen una cierta restricción, expresada en lo posible como parte de la invariante de B que no está incluida en la invariante de A. Cualquier característica introducida por B debería ser una consecuencia lógica de la restricción agregada. A y B deberían ser las dos diferidas o las dos efectivas.

Herencia por extensión: B introduce características no presentes en A y no aplicables a instancias directas de A. A debe ser efectiva. La extensión se aplica a las características y las restricciones a las instancias.

Herencia de variación funcional: B redefine algunas características de A y alguna de esas redefiniciones afectan el cuerpo de estas características, no solo con la signatura. A y B son ambas diferidas o ambas efectivas y B no debe introducir ninguna nueva característica excepto cuando es necesario.

Herencia de variación de tipo: B redefine algunas características de A y la redefinición afecta solo a signaturas. A y B son ambas diferidas o las dos efectivas y B no debe introducir ninguna nueva característica excepto cuando es necesario.

Herencia inefectiva: B redefine algunas de las características efectivas de A en características diferidas. Este tipo de herencia no es común porque va en la dirección contraria de la herencia. Uno normalmente espera que B sea más concreta y que A sea más abstracta. Usada en herencia múltiple y clases concretas reutilizadas.

Herencia de cosificación: A representa un tipo de estructura de datos y B representa una opción parcial o completa de implementación para la estructura de datos.

Herencia de estructura: A es una clase diferida que representa una propiedad estructural general y B que puede ser diferida o efectiva representa un cierto tipo de objeto que posee esa propiedad

Herencia de implementación: B obtiene de A un conjunto de características necesarias para implementar la abstracción asociada con B. A y B deben ser efectivas.

Herencia facilitada: A existe solamente para proveer un conjunto de características relacionadas lógicamente para el beneficio de sus herederos como B.