

Diseño Orientado a Objetos

Santa Tecla
parqueNaturalSantaTecla@gmail.com
Version 0.0.1

Índice

Justificación: ¿Por qué?

Definición: ¿Qué?

Teoría de Lenguajes

- Datos polimórficos

- Operaciones polimórficas

Objetivos: ¿Para qué?

Principio Abierto/Cerrado

Descripción: ¿Cómo?

Reusabilidad

- Herencia vs Parametrización

- Herencia vs Composición

Flexibilidad

- Clases Abstractas

- Interfaces

- Inversión de Control

Jerarquización

- Código Sucio por Herencia Rechazada

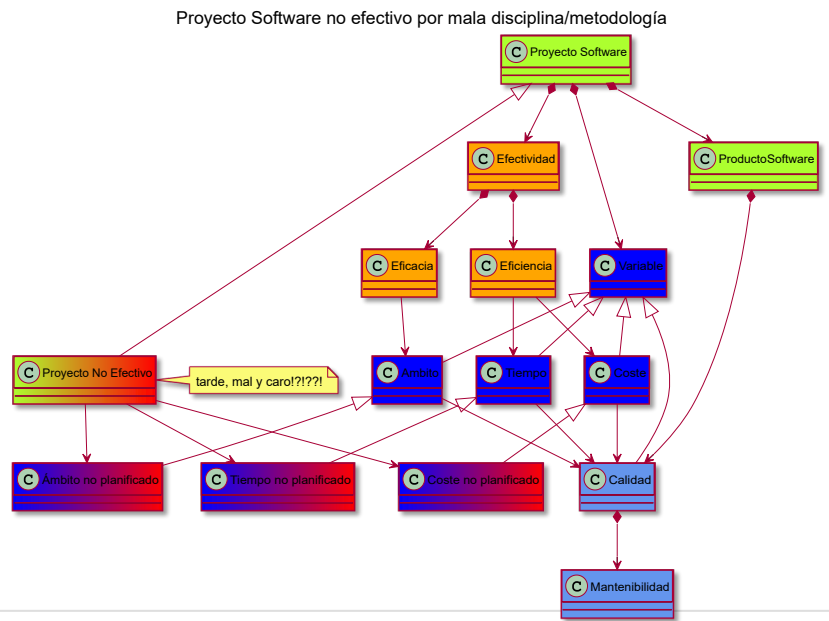
- Principio de Sustitución de Liskov

- Herencia vs Delegación

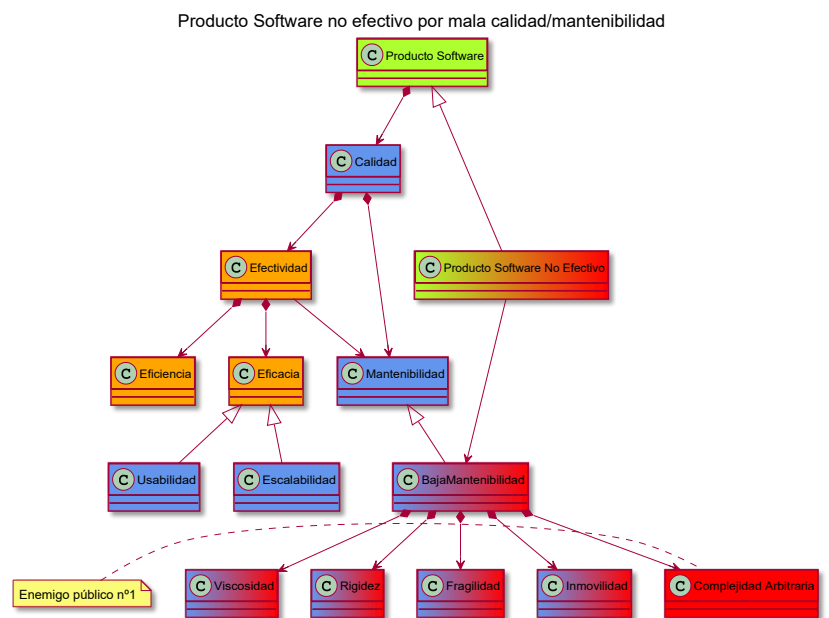
Bibliografía

Justificación: ¿Por qué?

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

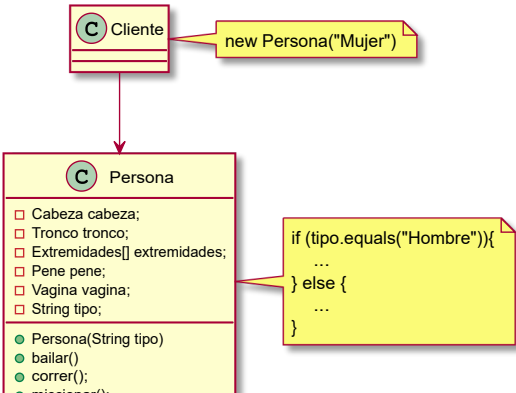
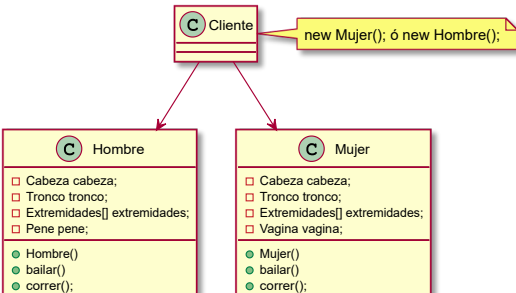
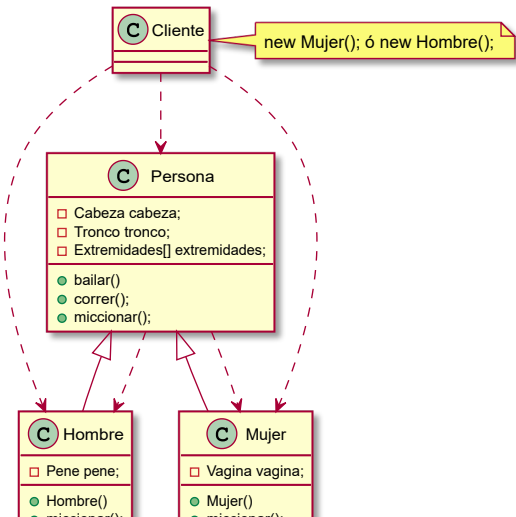


- Producto Software poco efectivo
 - porque tiene **mala calidad**
 - Poco **eficiente**
 - Poco eficaz en **corrección, usabilidad, escalabilidad, ...**
 - porque tiene **mala mantenibilidad**, de la que depende la eficiencia y la eficacia anteriores, porque es
 - **viscoso**, porque no se puede entender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



- El resultado del diseño modular **no asegura un código mantenible, de calidad** porque cuando hay distintos jerarquías de tipos de elementos existen dos posibles soluciones:

Reparto de Responsabilidad	Ejemplo	Problema de diseño modular

Reparto de Responsabilidad	Ejemplo	Problema de diseño modular
<p>Una única clase asume la responsabilidad de toda la jerarquía</p>		<ul style="list-style-type: none"> • Baja cohesión: incumpliendo el Principio de Única Responsabilidad • Clase grande: o propensa a ser grande con métodos largos cuando lleguen nuevos subtipos por la Ley del Cambio Continuo junto con "no hay 2 sin 3"
<p>Existe una clase por cada tipo de elemento que asumen sus correspondientes responsabilidades</p>		<ul style="list-style-type: none"> • Alto acoplamiento: de los clientes de la jerarquía porque conocen a todas las clases descendientes • DRY: con código repetidos en distintas clases a mantener, documentar, probar, ...
<p>Existe una jerarquía de clases por cada tipo de elemento que asumen sus correspondientes responsabilidades</p>		<ul style="list-style-type: none"> • Alto acoplamiento: de los clientes de la jerarquía porque conocen a todas las clases descendientes y existen ciclos entre las clases base y descendientes de la jerarquía, ...

Definición: ¿Qué?

“Mi conjetura es que la orientación a objetos será en los 80 lo que la programación estructurada en los 70. Todo el mundo estará a favor suyo. Cada productor prometerá que sus productos lo soportan. Cada director pagará con la boca pequeña el servirlo. Cada programador lo practicará. Y nadie sabrá exactamente lo que es!

— Rentsch

Object-Oriented Programming, SIGPLAN Notices vol. 17(12). 1982

“X es bueno. Orientado a Objetos es bueno. Ergo, X es Orientado a Objetos

— Stroustrup

The C++ Programming Language. 1988

- Basado en:



- Sistemas complejos
- Modelo del Dominio
- Legibilidad
- Diseño Modular

Sistemas complejos	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • Obtener la estructura de relaciones entre clases con buenas abstracciones e implementaciones mediante: <ul style="list-style-type: none"> ◦ 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	<ul style="list-style-type: none"> Somos escritores y respetamos: <ul style="list-style-type: none"> buenos nombres, comentarios y formato, estándares, consistencias y alarmas, DRY y código muerto, YAGNI, enfoque, al grano!
Diseño Modular	<ul style="list-style-type: none"> Todo módulo (método, clase y/o paquete) con <ul style="list-style-type: none"> Alta cohesión Bajo acoplamiento Tamaño pequeño

- Diseño Orientado a Objetos** incorpora dos mecanismos originarios de la Inteligencia Artificial (*frames*):

Mecanismo	Descripción
Herencia para aportar más reusabilidad	Transmisión de todo miembro, atributos y métodos, de una clase base a sus clases derivadas, como un <i>copy+paste dinámico</i> porque si cambio la clase base repercute a todas las clases derivadas
Polimorfismo para aportar más flexibilidad	Relajación del sistema de tipos donde la dirección de un objeto a una clase puede ser sustituida por la dirección a un objeto de cualquier clase derivada

Teoría de Lenguajes

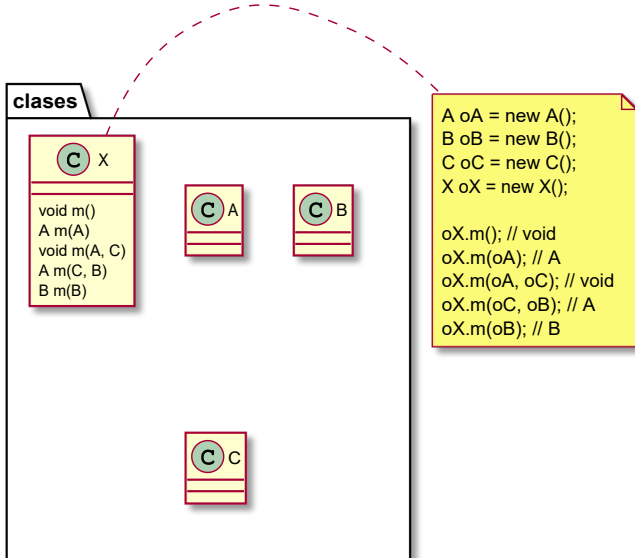
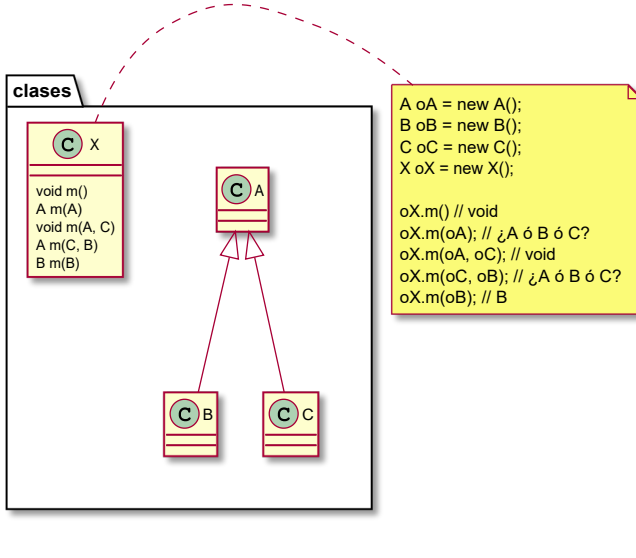
- Los **lenguajes de programación tienen diversos elementos**: clases, sentencia iterativa 0..N, un tipo primitivo, un cierre (closure), un parámetro, ... dependiendo del paradigma del lenguaje
- Cada **elemento tiene distintas características**: nombre, dirección, tamaño, ...
 - Por ejemplo:
 - un método tiene un nombre, una secuencia de parámetros (cada uno con sus características), un cuerpo y un valor de retorno
 - una constante tiene un nombre, un tipo, un valor, ...
 - una variable tiene un nombre, un tipo, un valor, una dirección, ...

Enlace estático	Enlace dinámico
enlace que se puede resolver en tiempo de compilación, característica que se puede determinar mirando el código	enlace que solo se puede resolver en tiempo de ejecución, característica que se puede determinar en un instante de la ejecución del código
<ul style="list-style-type: none"> • Ejemplos: <ul style="list-style-type: none"> ◦ final int MAX = 10; <ul style="list-style-type: none"> ▪ nombre: MAX // estático ▪ tipo: int // estático ▪ valor: 10 // estático ▪ ... ◦ int age; <ul style="list-style-type: none"> ▪ nombre: age // estático ▪ tipo: int // estático ▪ valor: ¿? // dinámico ▪ dirección: ¿? // dinámico ▪ ... 	<ul style="list-style-type: none"> • Ejemplos: <ul style="list-style-type: none"> ◦ final int MAX = 10; <ul style="list-style-type: none"> ▪ nombre: MAX // estático ▪ tipo: int // estático ▪ valor: 10 // estático ▪ ... ◦ int age; <ul style="list-style-type: none"> ▪ nombre: age // estático ▪ tipo: int // estático ▪ valor: 666 ó 0 ó ... // dinámico ▪ dirección: ¿0h a FFFFFFFFh? // dinámico ▪ ...

Datos polimórficos

- ¿Cuál es el tipo de enlace, estático o dinámico, entre una **expresión** (elemento del lenguaje) y el **tipo del valor del resultado de su evaluación** (característica del elemento del lenguaje)?
 - **Atención:** No se pregunta por "¿Cuál es el tipo de enlace, estático o dinámico, entre una expresión (elemento del lenguaje) y el valor del resultado de su evaluación (característica del elemento del lenguaje)?" cuya respuesta obviamente es un enlace dinámico!

Sin polimorfismo	Con polimorfismo

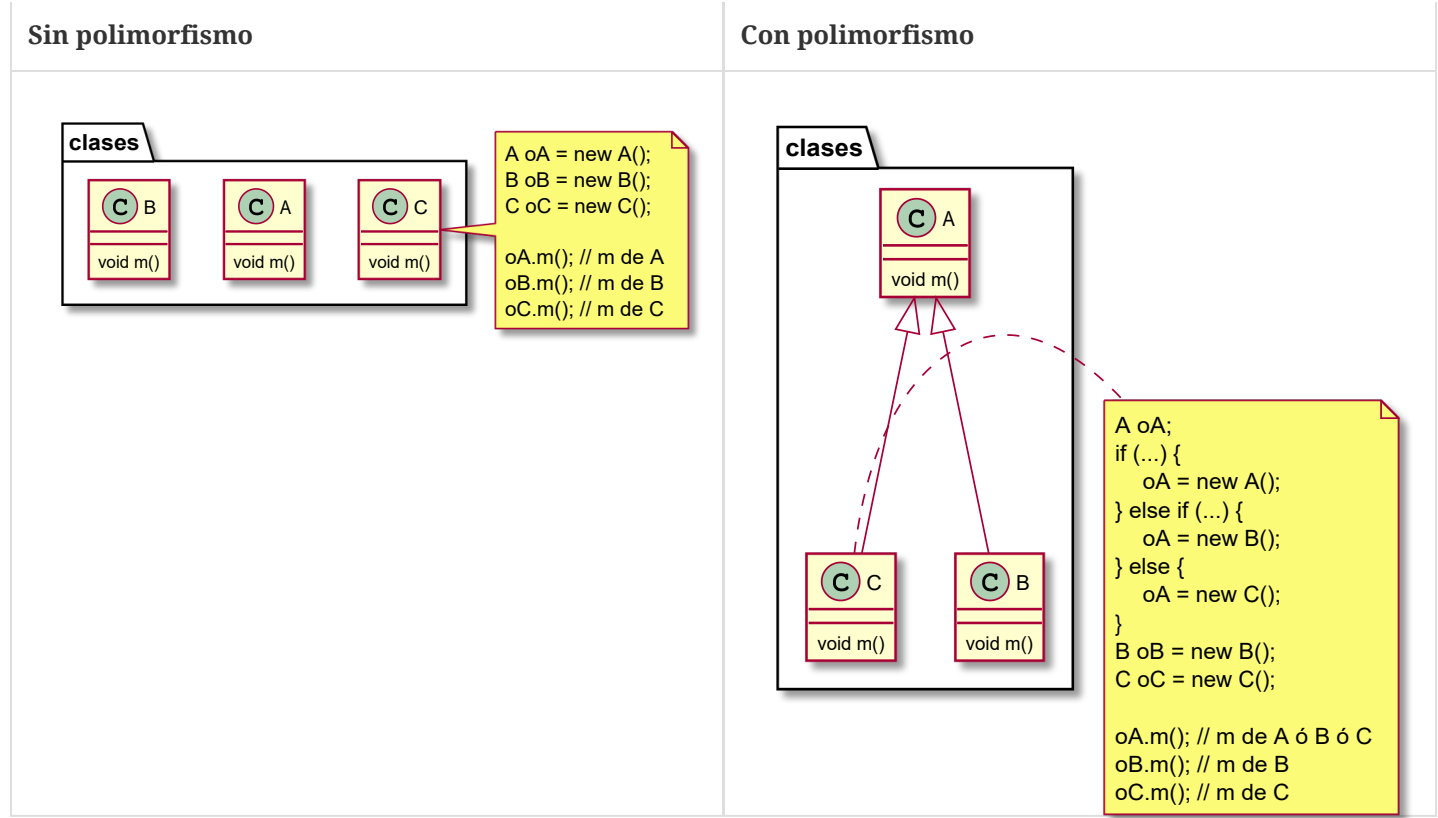
Sin polimorfismo	Con polimorfismo
Enlace estático, para toda expresión puede deducirse el tipo del valor del resultado de su evaluación en tiempo de compilación	Enlace dinámico, existen expresiones para las que no puede deducirse el tipo del valor del resultado de su evaluación en tiempo de compilación
Incluso con sobrecarga pero restringida para varios métodos con el mismo nombre y con diferentes parámetros, en número y/o tipos correspondientes, para evitar la ambigüedad con el tipo del valor de retorno	Con cualquier expresión que devuelva la dirección a un objeto declarada a una clase base , dado que el tipo del objeto referenciado puede ser de la clase base, si no es abstracta, o de cualquiera de sus descendientes, por la "relajación del sistema de tipos" del polimorfismo
	

- Por tanto, se define el **polimorfismo como enlace dinámico de expresiones al tipo devuelto por su evaluación**

Operaciones polimórficas

- ¿Cuál es el tipo de enlace, estático o dinámico, entre un **mensaje** (elemento del lenguaje) y el **método correspondiente a su ejecución** (característica del elemento del lenguaje)?

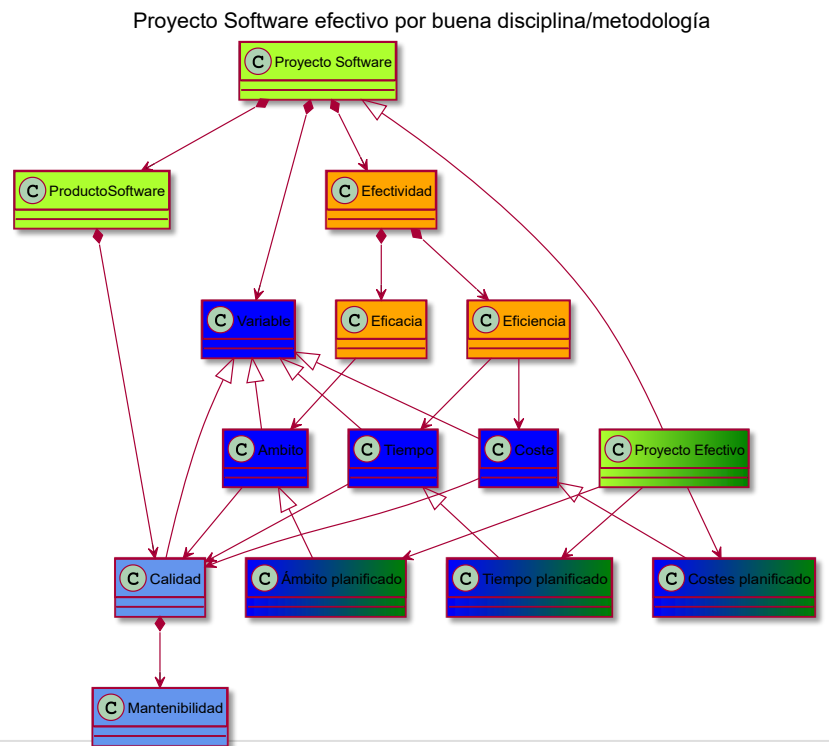
Sin polimorfismo	Con polimorfismo
Enlace estático, un mensaje lanzado a un objeto ejecutará el método con el mismo nombre y parámetros, en número y tipos correspondientes, de la clase del objeto	Enlace dinámico, un mensaje lanzado a un objeto polimórfico ejecutará un método con el mismo nombre y parámetros, en número y tipos correspondientes, de alguna de las clases de la jerarquía a partir de la clase base de la referencia
Incluso con sobrecarga pero restringida para varios métodos con los mismos parámetros, en número y tipos correspondientes, para evitar la ambigüedad con el tipo del valor de retorno	Con cualquier expresión que devuelva una referencia/puntero a una clase base , dado que el tipo del objeto referenciado puede ser de la clase base, si no es abstracta, o de cualquiera de sus descendientes , por la "relajación del sistema de tipos" del polimorfismo



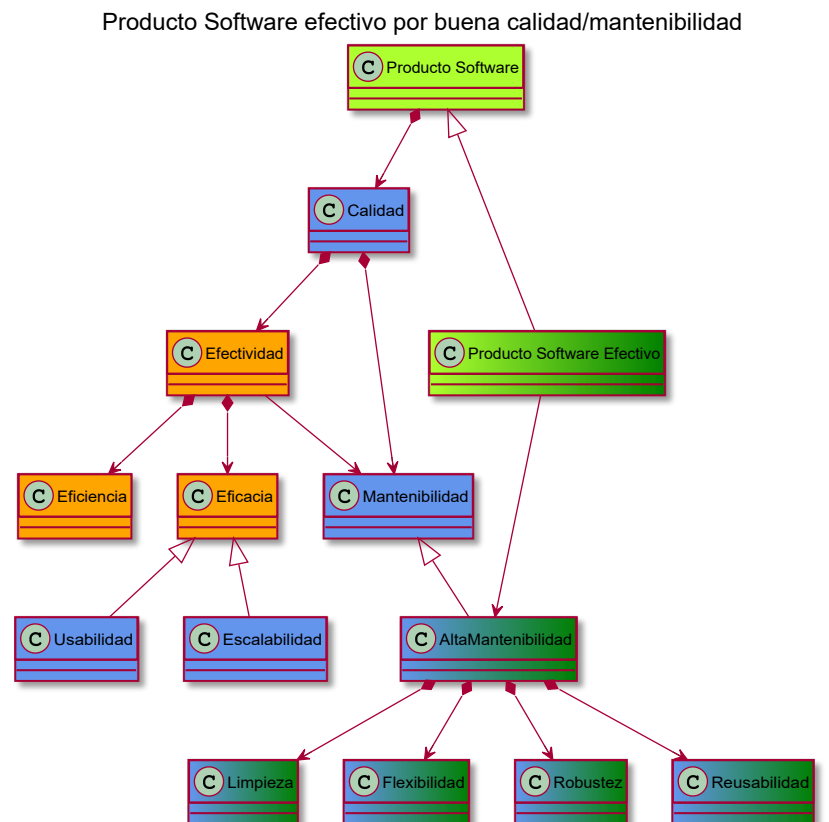
- Por tanto, se define el **polimorfismo como enlace dinámico de mensajes a métodos ejecutados**

Objetivos: ¿Para qué?

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



- 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
 - **fluido**, 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



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 y/o clasificación de clases pequeñas, sin ciclos!**

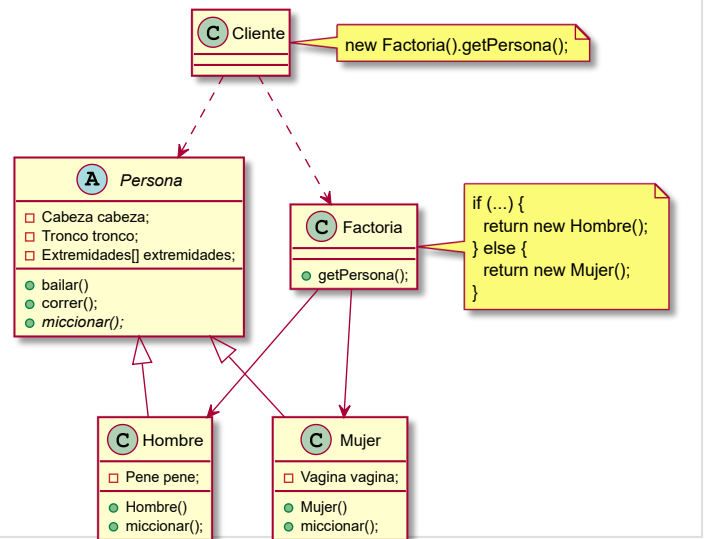
Flexible	Reparto de responsabilidades equilibrado y centralizado en clases que requiere modificarse únicamente si cambian los requisitos correspondientes, jerarquías de clases con alta cohesión, sin ciclos!
Resuable	Presencia de multitud de clases pequeñas, cohesivas y poco acopladas a tecnologías, algoritmos, ...!
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!

Principio Abierto/Cerrado

- Definido por **Bertran Meyer** (*Open/Close Principle* -OCP) en 1988 e incluido por **Robert Martin** como uno de los principios SOLID
- 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**, para que si hay un nuevo tipo de extensión no afecta a todos los clientes de la jerarquía y, así, no "romper" la versión actual que funciona: **Si no está roto, no lo toques!**



- Solución:**

- 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.
- Usando los principios de la programación orientada a objetos, **es posible** 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 todas las posibles clases derivadas**. Es posible 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 usar atributos que no sean privados**
- No usar variables globales**
- No preguntar por el tipo de objeto polimórfico**
- 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 completo, 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!**

Descripción: ¿Cómo?

Reusabilidad

- En Programación Orientada a Objetos existen **tres mecanismos** de reusabilidad:
 - Composición
 - Parametrización
 - Herencia

Herencia vs Parametrización

- La parametrización es para aquellos casos en que **la variabilidad se ciñe al tipo de elemento** que tratan entre varias clases
 - Por ejemplo: el código que diferencia las clases para
 - una lista de cerdos y una lista de deseos, se ciñe únicamente al tipo de elemento, cerdo o deseo
 - un distribuidor de tareas y un distribuidor de informes, se ciñe únicamente al tipo de elemento, tarea o informe

Lista de cerdos	Lista de deseos	Lista genérica
<pre> class Pig { ... } class PigPile { private Pig[] pigs; private int top; private int size; public PigPile(int size){ this.pigs = new Pig[size]; this.top = 0; } public void push(Pig pig){ assert pig != null; assert this.top+1 < this.pigs.length this.pigs[this.top] = pig; this.top++; } public Pig pop(){ assert this.top>0; this.top--; return this.pigs[this.top]; } public boolean empty(){ return this.top==0; } } PigPile pigs = new PigPile(3); </pre>	<pre> class Wish { ... } class WishPile { private Wish[] wishes; private int top; private int size; public WishPile(int size){ this.wishes = new Wish[size]; this.top = 0; } public void push(Wish wish){ assert wish != null; assert this.top+1 < this.wishes.length this.wishes[this.top] = wish; this.top++; } public Wish pop(){ assert this.top>0; this.top--; return this.wishes[this.top]; } public boolean empty(){ return this.top==0; } } WishPile wishes = new WishPile(1000); </pre>	<pre> class Pile<E> { private E[] items; private int top; private int size; public Pile(int size){ this.items = new E[size]; this.top = 0; } public void push(E item){ assert item != null; assert this.top+1 < this.items.length this.items[this.top] = item; this.top++; } public E pop(){ assert this.top>0; this.top--; return this.items[this.top]; } public boolean empty(){ return this.top==0; } } class Pig { ... } Pile<Pig> pigs = new Pile<Pig> (3); class Wish { ... } Pile<Wish> wishes = new Pile<Wish>(1000); </pre>

- Con la **parametrización o genericidad** se evita **re-codificar, re-probar, re-documentar, ..., re-mantener** todas las clases cuando hay nuevos tipos de listas o cuando hay que modificar el comportamiento de todas las pilas por un diseño alternativo o error

Herencia vs Composición

Composición	Herencia
Reusabilidad por ensamblado	Reusabilidad por extensión
<pre> class Parte { public Parte(); public void m1(); public void m2(); public void m3(); } class Todo { private Parte parte; public Todo() { this.parte = new Parte(); } public void m4(){ ... this.parte.m1(); this.parte.m3(); ... } public void m5(){ ... } } </pre> <p style="text-align: right;">JAVA</p>	<pre> class Base { public Base(); public void m1(); public void m2(); public void m3(); } class Descendiente extends Base { public Descendiente() { super(); } public void m4(){ ... this.m1(); this.m3(); ... } public void m5(){ ... } public void m1(){ ... super.m1(); ... } public void m2(){ ... } } </pre> <p style="text-align: right;">JAVA</p>
Reusabilidad con desarrollo explícito en el código , declarando los atributos que son parte del todo y enviando mensajes para su gestión	Reusabilidad implícita en el lenguaje , declarando la herencia se transmiten automáticamente todos los atributos y métodos, públicos, protegidos, privados y de paquete, con unos accesos u otros dependiendo del punto de vista (clase cliente o clase descendiente)
Más objetos para la reusabilidad, se tiene un objeto para el todo y otro para la parte y, por tanto, menos eficiente por la re-emisión de mensajes del objeto "todo" al objeto "parte"	Menos objetos para la reusabilidad, se tiene un objeto de la clase descendiente y, por tanto, más eficiente por la emisión directa de mensajes al objeto "descendiente"
Relación dinámica entre objetos, por tanto es más flexible porque un todo puede colaborar con distintas partes en el tiempo creando nuevos objetos	Relación estática entre clases, por tanto es menos flexible porque un objeto de la clase descendiente es y será un objeto de la clase descendiente sin poder modificar su clase base en tiempo de ejecución

Composición	Herencia
Caja negra , desde la clase todo se tiene acceso a miembros públicos de la parte, sin posibilidad de modificar el código reusado	Caja blanca , desde la clase descendiente se tiene acceso a miembros públicos y protegidos y de paquete, con posibilidad de modificar el código reusado mediante la redefinición (<i>@Override</i>)
Fácil de mantener porque es imposible romper el principio de encapsulación	Difícil de mantener porque es fácil romper el principio de encapsulación

“Favorecer la composición de objetos frente a la herencia de clases

— Gamma et al
Patrones de Diseño

- solo usar jerarquías de herencia cuando sean muy sencillas, limpias, claras, ... sin chapuzas!

Ley Flexible y Estricta de Demeter

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

- **Justificación:** Controlar el bajo acoplamiento restringiendo a qué objetos enviar mensajes desde un método

Ley estricta de Demeter	Ley flexible de Demeter
<ul style="list-style-type: none"> • Enviar mensajes únicamente a: <ul style="list-style-type: none"> ◦ <i>this</i> y <i>super</i> ◦ Atributos de la clase ◦ Parámetro del método ◦ Local del método ◦ No enviar nunca a otros objetos indirectos obtenidos como resultado de un mensaje a un objeto de conocimiento directo. 	<ul style="list-style-type: none"> • Enviar mensajes únicamente a: <ul style="list-style-type: none"> ◦ <i>this</i> y <i>super</i> <ul style="list-style-type: none"> ■ Atributos de la clase y de la clase base ■ Parámetro del método ■ Local del método ■ No enviar nunca a otros objetos indirectos obtenidos como resultado de un mensaje a un objeto de conocimiento directo.

Flexibilidad

Clases Abstractas

- Aquellas clases que **no son instanciables** porque tienen algún **método abstracto**, sin definición, lo cuál imposibilita su instanciación ante la ejecución de mensajes correspondientes a métodos abstractos sin definición
- **Facilitan la reusabilidad**, de todos los atributos y métodos definidos
- **Facilitan la flexibilidad** mediante el polimorfismo que aporta una relajación del sistema de tipos sobre una jerarquía de herencia
 - **Sin la clase base abstracta no hay herencia**, no hay posible polimorfismo de datos
 - **Sin el método abstracto no hay interfaz**, no hay posible polimorfismo de operaciones

Patrón Método Plantilla

Sinónimos	Synonyms	Libro	Autor
Método Plantilla	<i>Template Method</i>	Patrones 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 cambiarla estructura del algoritmo de la clase padre

Sin método plantilla	Con método plantilla
<pre> class X { } class Y extends X { public void m() { // aaaaaaaaaaaaa // yyyyyyyyyyyyyy // bbbbbbbbbbbbbb } } class Z extends X { public void m() { // aaaaaaaaaaaaa // zzzzzzzzzzzzz // bbbbbbbbbbbbbb } } </pre>	<pre> class X { public void m() { // aaaaaaaaaaaaa this.middle(); // bbbbbbbbbbbbbb } public abstract middle(); } class Y extends X { public middle(){ // yyyyyyyyyyyyyy } } class Z extends X { public middle(){ // zzzzzzzzzzzzz } } </pre>

Interfaces

- Son **clases abstractas puras**, sin definición de atributos y métodos, solo cabeceras de los métodos abstractos, la interfaz de la clase

- No aportan reusabilidad pero
- **Facilitan la flexibilidad** mediante el polimorfismo que aporta una relajación del sistema de tipos sobre una jerarquía de herencia
 - **Sin la clase base abstracta pura no hay herencia**, no hay posible polimorfismo de datos
 - **Sin los métodos abstractos no hay interfaz**, no hay posible polimorfismo de operaciones

Principio de Inversión de Dependencias

- Definido por **Robert Martin** (*Dependency Inversion Principle, DIP*) como uno de los principios SOLID, pudiendose entender como el resultado de aplicar rigurosamente los **Principios de Sustitución** de **Barbara Liskov** y **Abierto/Cerrado** de **Bertrand Meyer**

“*Los módulos de alto nivel no deberían depender de los módulos de bajo nivel. Ambos deberían depender de abstracciones*

— Robert Martin

Principio de Inversión de Dependencias

- Por ejemplo: en vez de que un Copier (modulo de alto nivel) lea de un KeyboardReader y escriba en PrintWriter (módulos de bajo nivel), debería leer de una interfaz Reader, base de KeyboardReader y escribir en una interfaz Writer, base de PrintWriter de tal forma que Copier, KeyboardReader y PrintWriter dependan de las abstracciones Reader y Writer

Sin Principio de Inversión de Dependencias	Con Principio de Inversión de Dependencias
<pre> class KeyboardReader{ public char[] read(){ ... } } class PrintWriter{ public write(char[]){ ... } } class Copier { public Copier(KeyboardReader reader, PrintWriter writer){ ... char[] data = reader.read(); ... writer.write(data); ... } } </pre> <p>JAVA</p>	<pre> interface Reader { char[] read(); } class KeyboardReader implements Reader{ public char[] read(){ ... } } interface Writer { void write(char[]); } class PrintWriter implements Writer{ public write(char[]){ ... } } class Copier { public Copier(Reader reader, Writer writer){ ... char[] data = reader.read(); ... writer.write(data); ... } } </pre> <p>JAVA</p>

- 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 los objetos de bajo nivel dentro de la clase de alto nivel no puede ser hecha con el operador new**

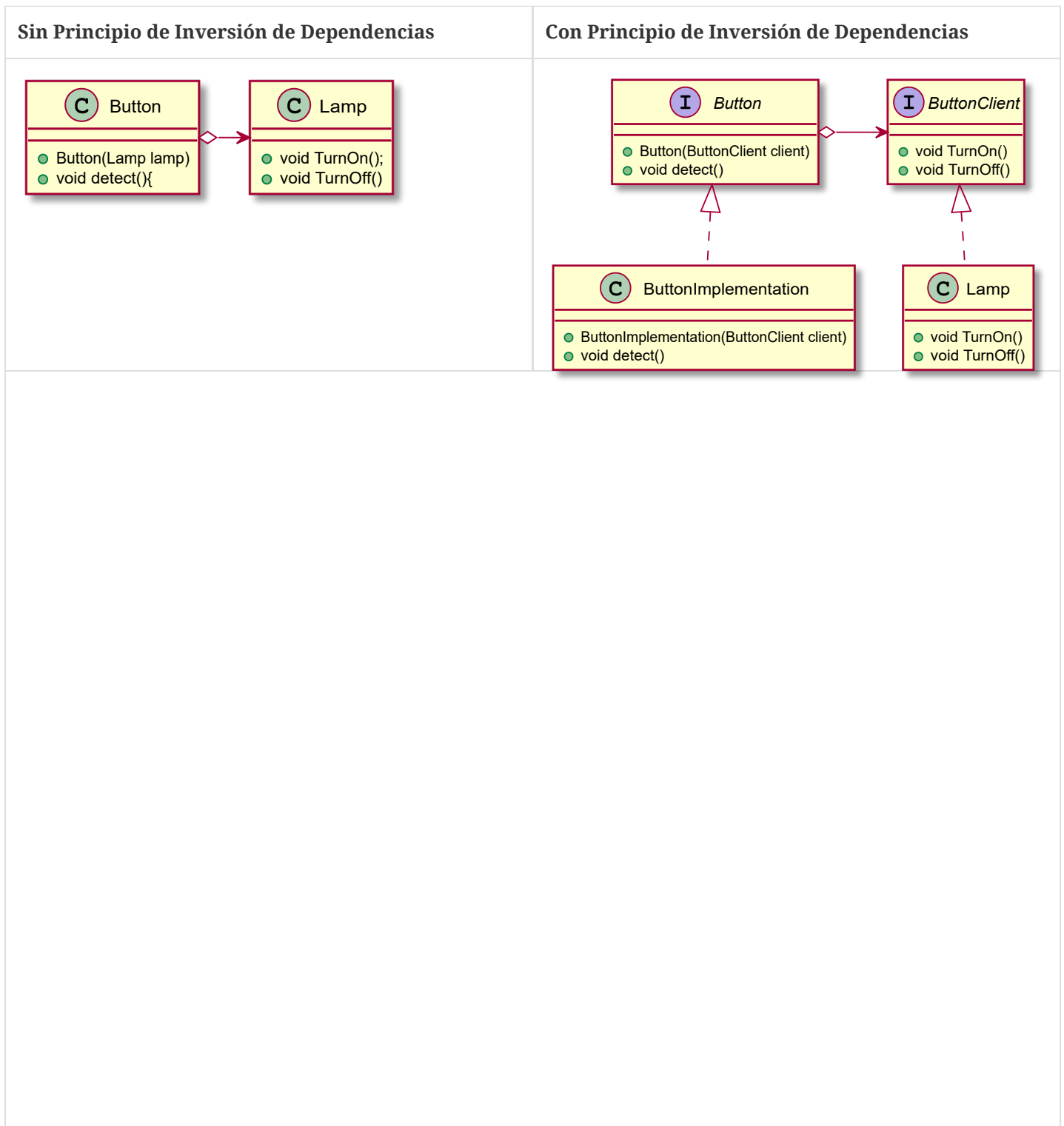
- Por ejemplo: la lógica de Copier será reutilizada sin cambios cuando nuevos dispositivos de entrada y salida entren en juego con el cambio de requisitos heredando de las interfaces Reader y Writer.

“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

— Robert Martin

Principio de Inversión de Dependencias

- Por ejemplo: en vez de que un Button dependa de una Lamp para encenderla y apagarla, un Button depende de un ButtonClient que puede encenderse y apagarse podrá reutilizarse por cualquier botón, ButtonImplementation, ... que herede de Button para encender y apagar cualquier cliente, Lamp, ... que herede de ButtonClient.



Sin Principio de Inversión de Dependencias	Con Principio de Inversión de Dependencias
<pre> class Lamp { public void TurnOn(){ ... } public void TurnOff(){ ... } } class Button { private Lamp lamp; public Button(Lamp lamp){ this.lamp = lamp; } void detect(){ boolean buttonOn = this.getPhysicalState(); if (buttonOn) { this.lamp.turnOn(); } else { this.lamp.turnOff(); } } boolean getPhysicalState(){ } } </pre>	<pre> interface ButtonClient { void TurnOn(); void TurnOff(); } class Lamp extends ButtonClient { public void TurnOn(){ ... } public void TurnOff(){ ... } } interface Button { void detect(); } class ButtonImplementation implements Button { private ButtonClient client; public ButtonImplementation(ButtonClient client){ this.client = client; } void detect(){ boolean buttonOn = this.getPhysicalState(); if (buttonOn) { this.client.turnOn(); } else { this.client.turnOff(); } } boolean getPhysicalState(){ } } </pre>

- 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**
 - Por ejemplo: 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*.
- **Contraindicaciones:** 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 aplicable 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!**

Principio 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 métodos únicamente por el beneficio de una de sus subclases**. Esta práctica es **indeseable** por que cada vez que una clase derivada necesite un nuevo método, éste será añadido a la clase base. Esto va a contaminar aún más la interfaz de la clase base, por lo que sería poco cohesiva.

- 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

“*Los clientes no deberían forzarse a depender de interfaces que no usan*

— Robert Martin

Principio de Separación de Interfaces

• Solución:

- Sería deseable evitar el acoplamiento entre clientes como sea posible y separar interfaces como sea posible. Dado que los clientes están “separados”, **las interfaces deben permanecer también “separadas”**.
- 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**.

Sin interfaces	Con interfaces
<pre> class Secretaria { public void setCalificación(int id, int calificacion){ ... } public int getIngresosFamiliares(int id){ ... } } class Alumno { public void matricular(Secretaria secretaria){ secretaria.setCalificación(10); } } class Profesor { public void calificar(Secretaria secretaria){ secretaria.getIngresosFamiliares(666); } } </pre>	<pre> interface SecretariaAlumnos { void setCalificación(int id, int calificacion); } interface SecretariaProfesores { int getIngresosFamiliares(int id); } class Secretaria implements SecretariaAlumnos, SecretariaProfesores { public void setCalificación(int id, int calificacion){ ... } public int getIngresosFamiliares(int id){ ... } } class Alumno { public void matricular(SecretariaAlumnos secretaria){ secretaria.setCalificación(10); // ERROR!!! } } class Profesor { public void calificar(SecretariaProfesores secretaria){ secretaria.getIngresosFamiliares(666); /// ERROR!!! } } </pre>

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

Inversión de Control

Sinónimos	Synonyms	Libro	Autor
Principio Hollywood: "No me llames, ya te llamaremos"	Hollywood Principle: "Don't call me, we'll call you"		

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

—Johanson & Foote
1988

- **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.**

“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”

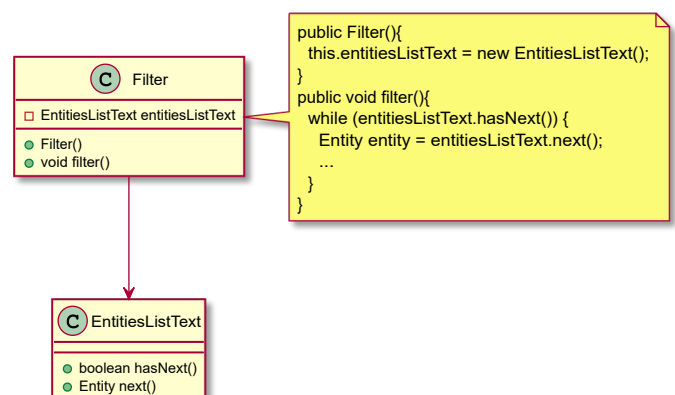
- **Variaciones:**

- **Patrón Método Plantilla** con redefinición de métodos abstractos invocados desde el método plantilla de la clase base
- **Eventos** con auditores que determinan su comportamiento
- **Configuración** con datos externos al framework para determinar el comportamiento
- **Inyección de Dependencias**

Inyección de Dependencias

Sinónimos	Synonyms	Libro	Autor
Inyección de Dependencias	<i>Pluggin</i>		
Patrón de Diseño Estrategia	<i>Strategy Pattern Design</i>	Patrones de Diseño	<i>Gamma et al</i>

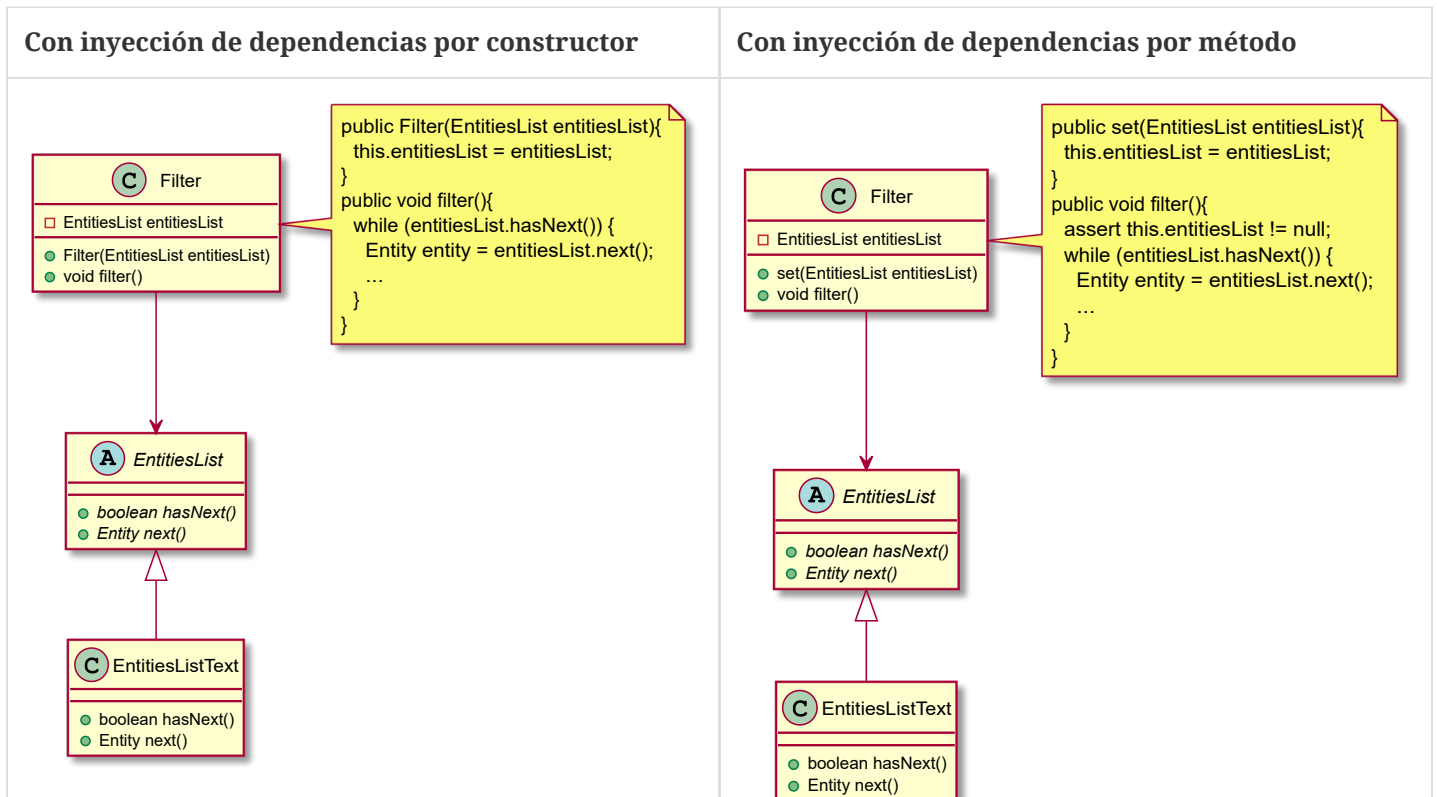
Justificación: Eliminar las dependencias de una clase de aplicación hacia la implementación de otra clase, servicio, para que esta clase sea reutilizada por implementaciones alternativas del servicio actual



- **Solución:**

- 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 instancie un objeto concreto para obtener el listado
 - *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 segundo será **inyectar 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 para la inyección del objeto será:
 - **por constructor**, muy recomendable 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** 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.



Jerarquización

- **Contexto:**

- La Relación de Composición responde a **A tiene un B**
- La Relación de Herencia responde a **A es un B**, del que surge el famoso acrónimo **ISA**

- **La posible 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*
- Ante la duda, si la cardinalidad de la parte/base en cuestión puede ser mayor que 1, **decantarse por la composición**

Tipo de herencia	Descripción	Ejemplo
Herencia por especialización	donde la clase descendiente implementa todas las operaciones de la clase base, añadiendo o redefiniendo partes especializadas	<i>Ingeniero de Sistemas es descendiente de Ingeniero añadiendo nuevos métodos</i>
Herencia por limitación	donde la clase descendiente no implementa todas las operaciones de la clase base, completamente desaconsejada porque imposibilita el tratamiento polimórfico	<i>Pingüino es descendiente de Ave con el método volar</i>
Herencia por construcción	donde realmente es una relación de composición, completamente desaconsejada si no existe herencia privada como en C++	<i>Por ejemplo: la clase Motor es la clase base de la clase Coche, un coche es un motor con puertas</i>
Herencia por extensión	donde la especialización transforma el concepto de la clase base a la clase derivada	<i>Por ejemplo: la clase Fracción es la clase base de la clase NodoFracción, es una fracción capaz de engancharse y desengancharse de otro nodo fracción</i>

- Siempre que se analiza/diseña una relación de herencia **se puede analizar/diseñar su contrapartida como relación de composición, por delegación**

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 delegación** para evitar corromper la interfaz de la clase derivada

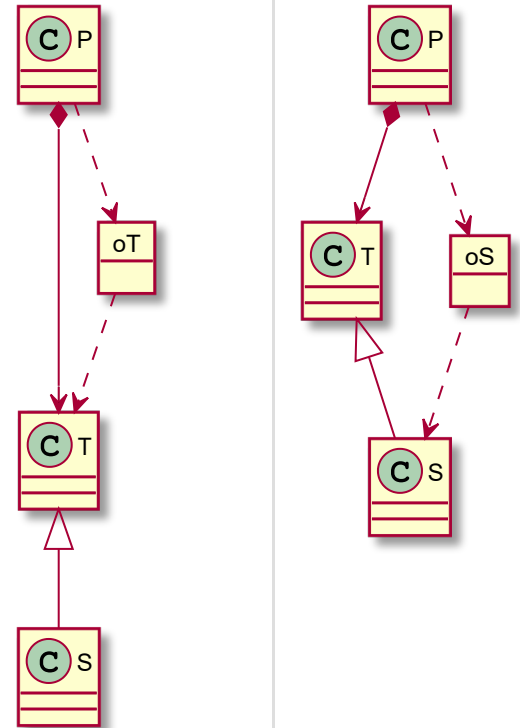
Principio de Sustitución de Liskov

- Definido por **Barbara Liskov** e incorporado por **Robert Martin** (*Liskov's Substitution Principle -LSP*) como uno de los principios **SOLID**

“Lo que se quiere aquí es algo como la siguiente propiedad de sustitución: si para cada objeto oT de un tipo T , hay un objeto oS de tipo S tal que para todo programa P definido en términos de T , el comportamiento de P no cambia cuando oT es sustituido por oS , entonces S es un subtipo de T ”

— Barbara Liskov

A behavioral notion of subtyping. ACM Transactions on Programming Languages and Systems (TOPLAS). Volume 16. Issue 6 (November 1994). pp. 1811. 1841



- 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**
- Por tanto, la relación de herencia se refiere al comportamiento. **No al comportamiento privado intrínseco si no al comportamiento público extrínseco del que dependen los clientes**

“Se cumple cuando se redefine un método en una derivada **reemplazando su precondition por una más débil** y su **postcondicion por una más fuerte**”

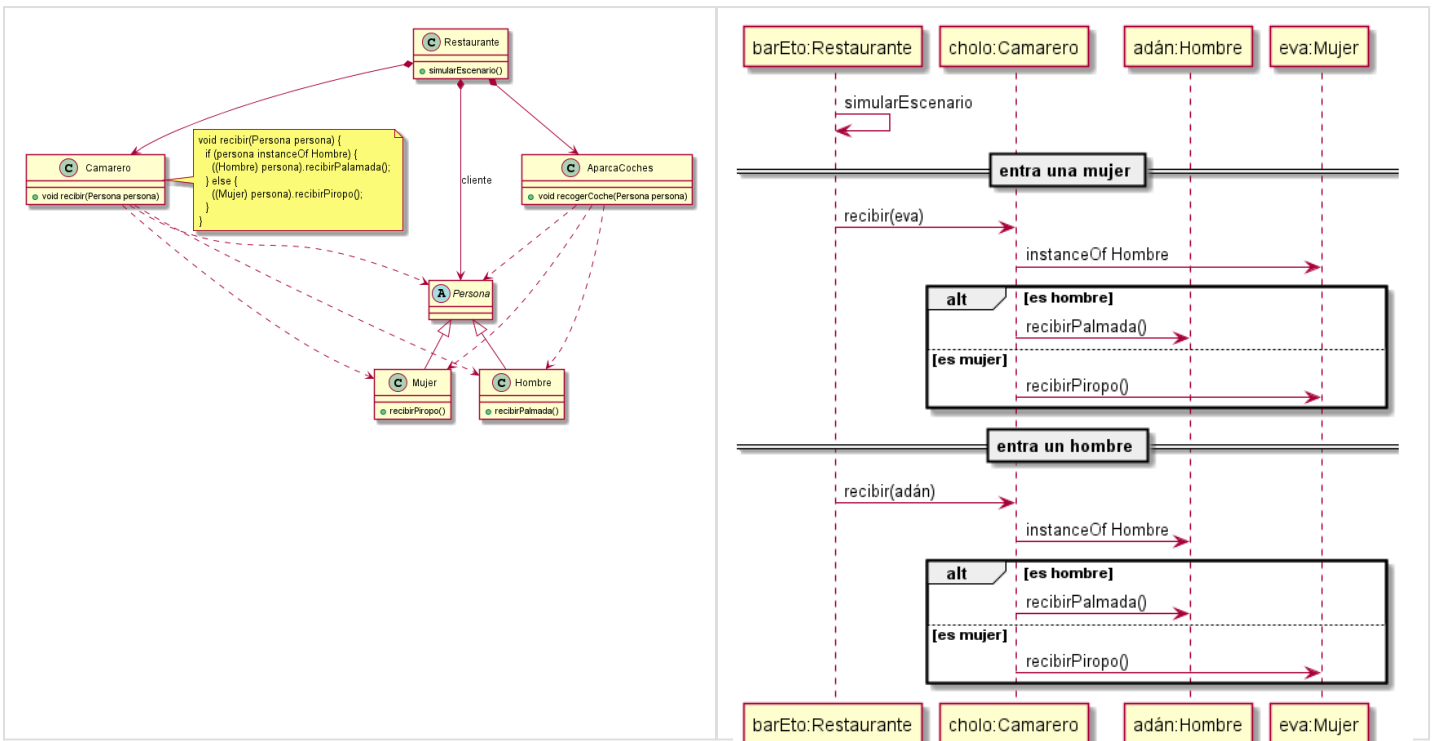
— Barbara Liskov
Principio de Sustitución

- Condiciones:
 - La precondition de un subtipo es creada combinando con el operador **OR** las precondition es del tipo base y del subtipo, lo que resulta una **precondition 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**.
- Violaciones:

- Una de las violaciones más evidentes de este principio es el uso de la Información de Tipos en Tiempo de Ejecución (*instanceof*, RTTI, ...) para seleccionar una función basada en el tipo de un objeto. Muchos ven esta estructura como el anatema de la Programación 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:
 - 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.

Técnica de Doble Despacho

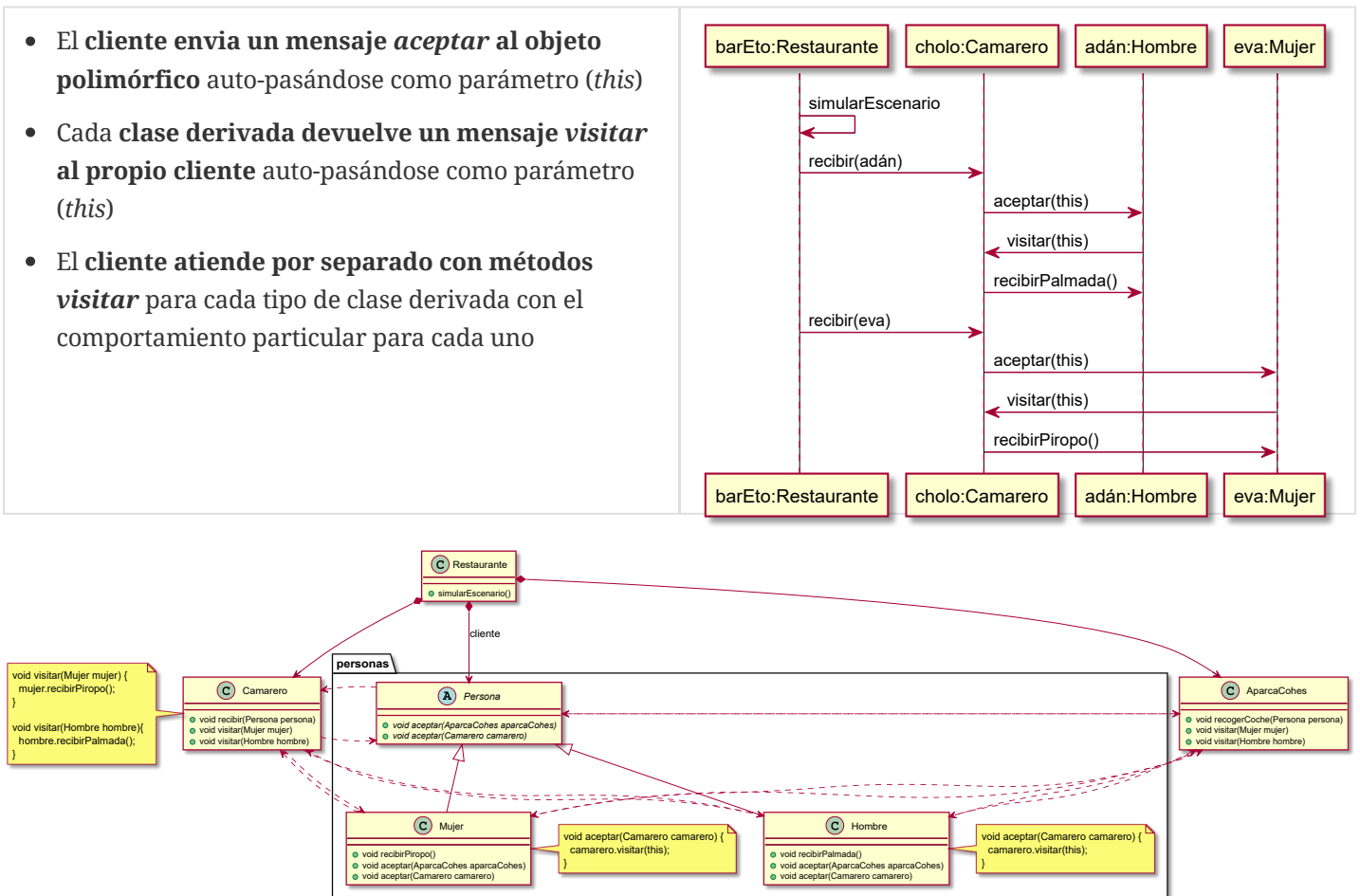
- Motivación:** Por un reparto de responsabilidades justificado, existe la necesidad de que un cliente de una jerarquía de clases **trate específicamente según la clase derivada concreta** de un parámetro polimórfico
 - Por ejemplo:
 - la secretaría de alumnos atiende de forma distinta para la matriculación de distintos tipos de alumnos: master en Cloud Apps, master en Ingeniería Web, ..., grados, ESA (para adultos), Erasmus (del Espacio Europeo), invitado, ...; un profesor para evaluar con distintos tipos de pruebas según el tipo de alumno concreto; ...
 - un camarero atiende de forma distinta para la recepción de distintos tipos de clientes: mujer, hombre, ...; un aparcacoches para recoger y entregar el coche según el tipo de persona; ...
- 1ª solución: preguntando por el tipo del objeto polimórfico (ver código)**
(<https://github.com/miw-upm/IWVG/tree/master/doo/src/main/java/dobleDespacho/v1/mal>)
 - De forma directa con operadores y funciones del lenguaje, *instanceOf*, o indirectamente con métodos explícitos, *get<Tipo>()*, o ... abriendo distintas ramas de sentencias alternativas para tratar cada tipo de clase derivada



- Consecuencias de la 1ª solución:**
 - viola el Principio de Sustitución de Liskov** preguntando por el tipo de objeto polimórfico
 - incurre en cambios divergentes** para atender con una nueva rama en cada clase cliente que hay que localizar por toda la aplicación

- **rompe el principio Open/Close** con cambios en el interior de los métodos del cliente
- **2ª solución: aplicando la técnica de doble despacho** (ver código)

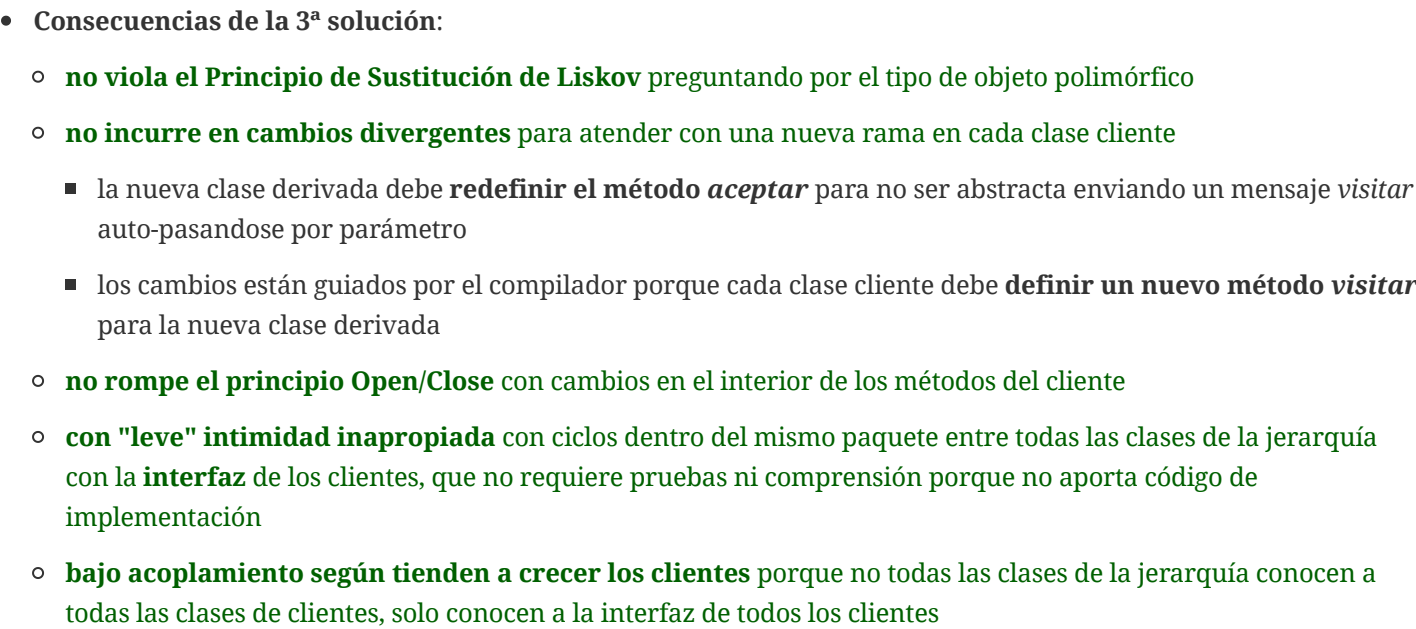
(<https://github.com/miw-upm/IWVG/tree/master/doo/src/main/java/dobleDespacho/v2/basic>)



- **Consecuencias de la 2ª solución:**
 - **no viola el Principio de Sustitución de Liskov** preguntando por el tipo de objeto polimórfico
 - **no incurre en cambios divergentes** para atender con una nueva rama en cada clase cliente
 - la nueva clase derivada debe **redefinir el método aceptar** para no ser abstracta enviando un mensaje *visitar* auto-pasandose por parámetro
 - los cambios están guiados por el compilador porque cada clase cliente debe **definir un nuevo método visitar** para la nueva clase derivada
 - **no rompe el principio Open/Close** con cambios en el interior de los métodos del cliente
 - **intimidad inapropiada** con ciclos entre todas las clases cliente con todas las clases de la jerarquía
 - **alto acoplamiento según tienden a crecer los clientes** porque todas las clases de la jerarquía conocen a todas las clases de clientes
- **3ª solución: Principio de Inversión de Dependencias** (ver código)

(<https://github.com/miw-upm/IWVG/tree/master/doo/src/main/java/dobleDespacho/v3/extensible>)

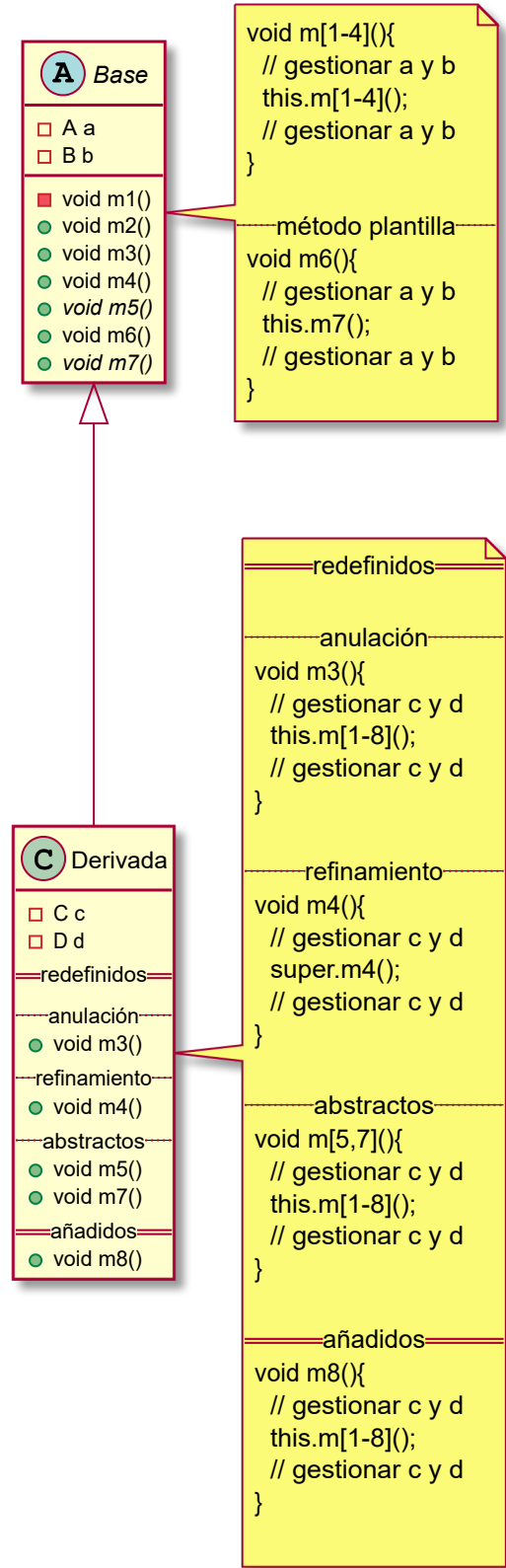
- La jerarquía de clases no conoce directamente a los clientes sino que conoce únicamente a una interfaz que cumple todo cliente que visita la jerarquía, *visitador* genérico



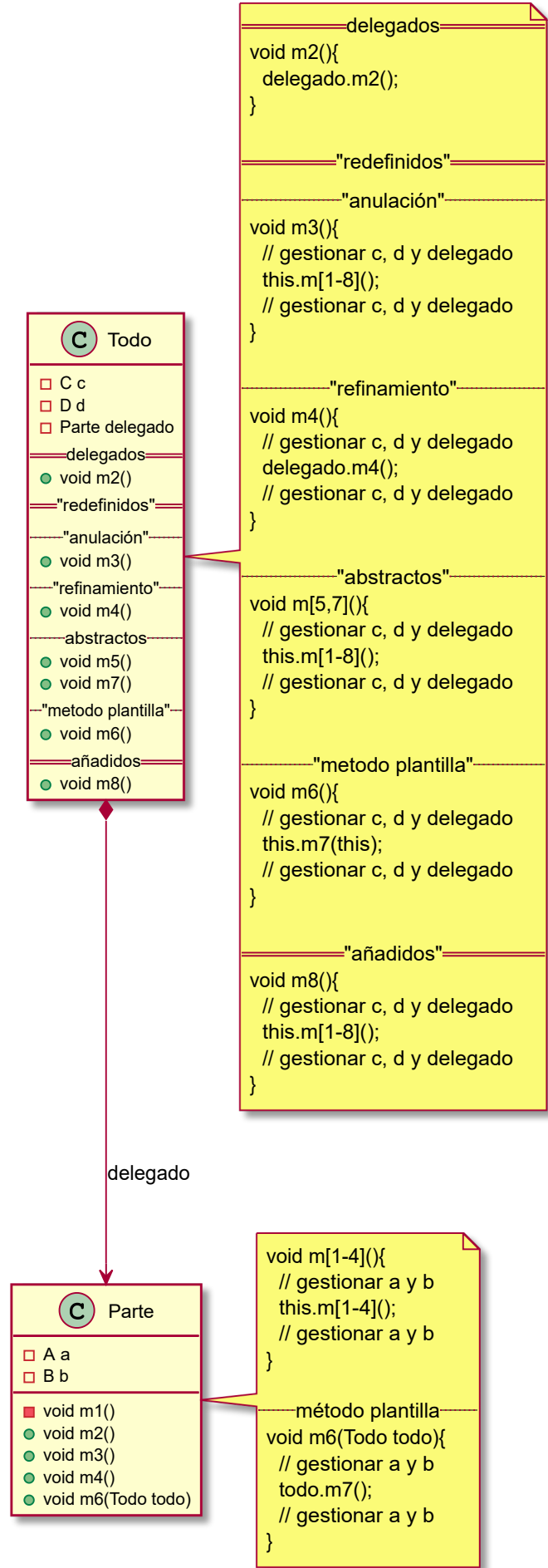
- Cualquier relación de herencia puede convertirse en una relación de composición

Relación de Herencia	Relación de Composición para Delegación

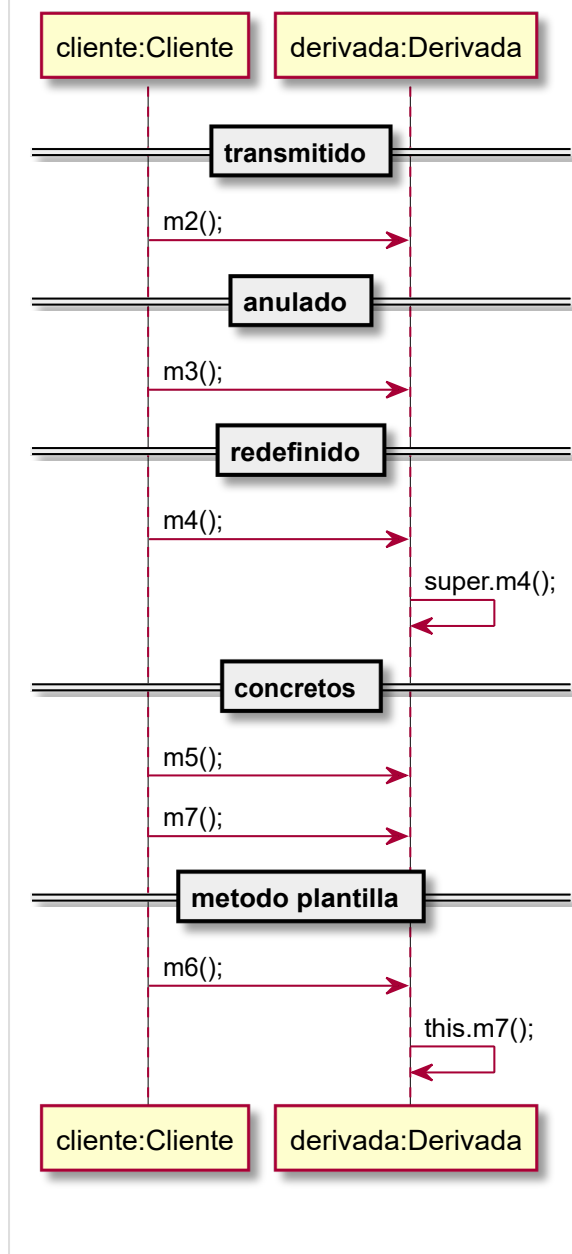
Relación de Herencia



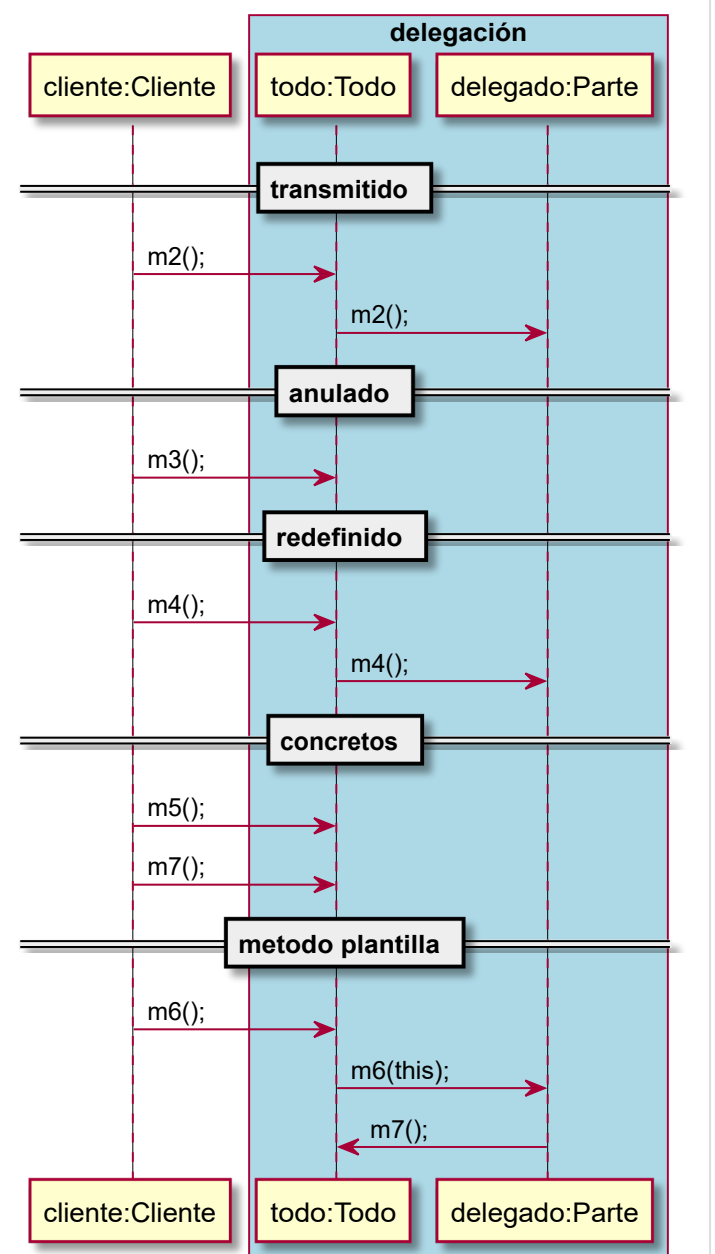
Relación de Composición para Delegación



Relación de Herencia



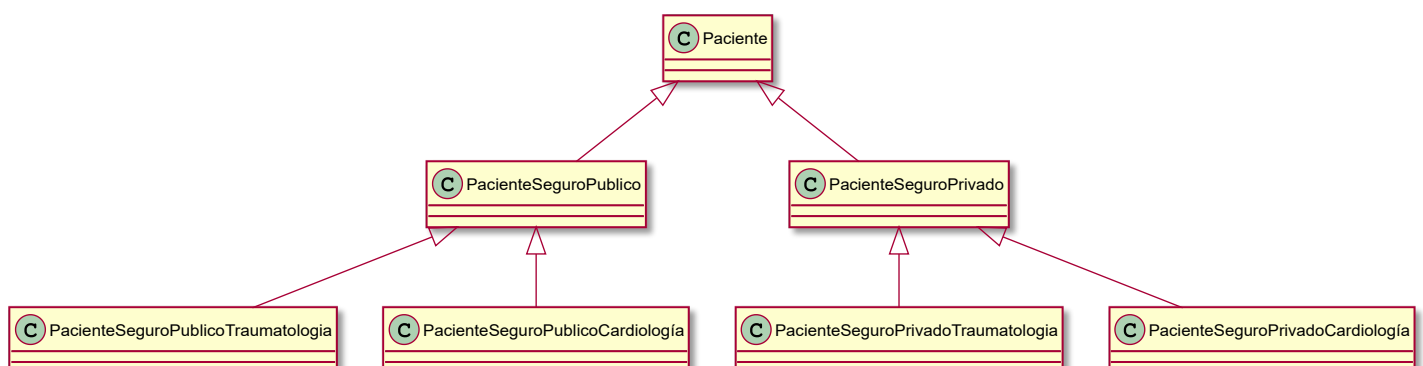
Relación de Composición para Delegación



Código Sucio por Jerarquías Paralelas de Herencia

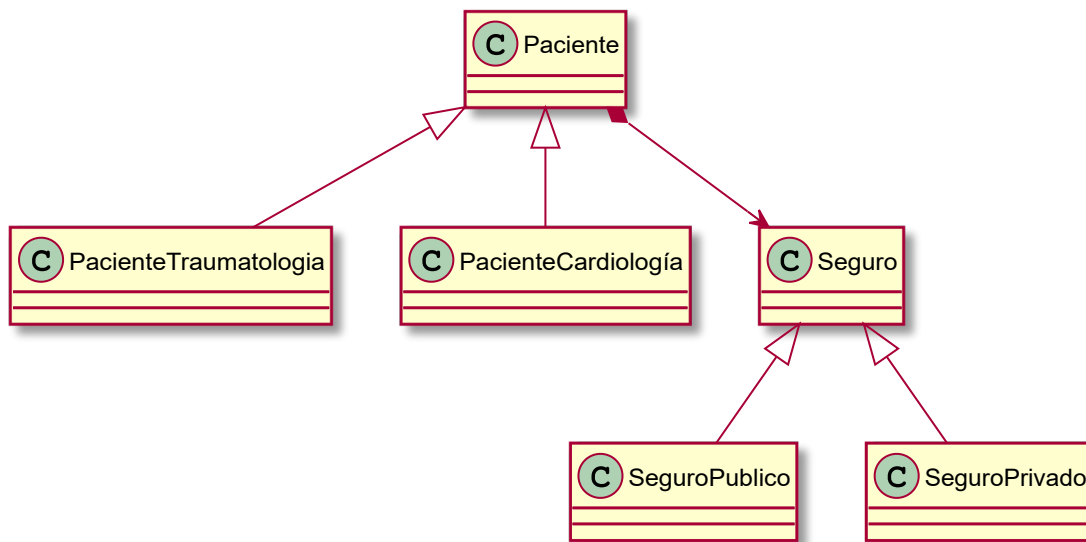
• Justificación:

- 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


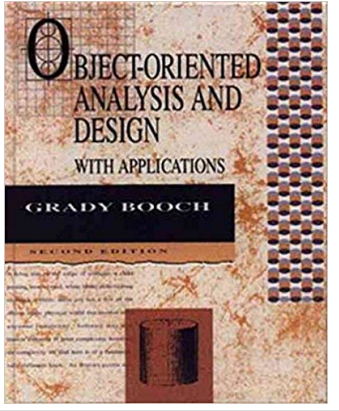
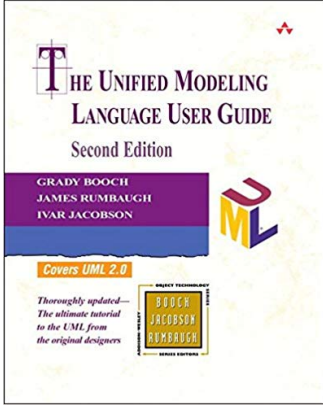
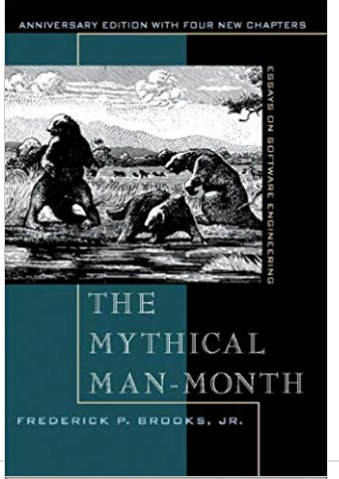
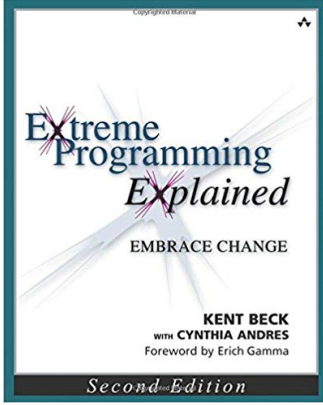
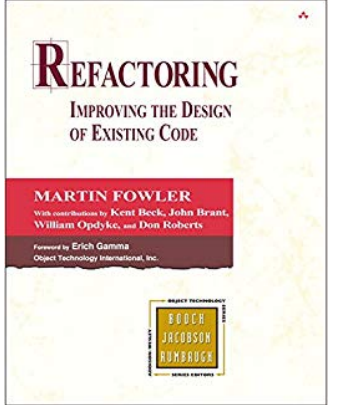


- **Solución:**

- 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



Bibliografía

Obra, Autor y Edición	Portada	Obra, Autor y Edición	Portada
<ul style="list-style-type: none"> Object Solutions. Managing the Object Oriented Project <ul style="list-style-type: none"> Grady Booch Addison-Wesley Professional (1789) 		<ul style="list-style-type: none"> Object Oriented Analysis and Design with Applications <ul style="list-style-type: none"> Grady Booch Imprint Addison-Wesley Educational s Inc (3 de junio de 2011) 	
<ul style="list-style-type: none"> The Unified Modeling Language User Guide <ul style="list-style-type: none"> Grady Booch Pearson Education (US); Edición: 2 ed (28 de junio de 2005) 		<ul style="list-style-type: none"> The Mythical Man Month. Essays on Software Engineering <ul style="list-style-type: none"> Frederick P. Brooks Prentice Hall; Edición: Nachdr. 20th Anniversary (1 de enero de 1995) 	
<ul style="list-style-type: none"> Extreme Programming Explained. Embrace Change. Embracing Change <ul style="list-style-type: none"> Kent Beck, Cynthia Andres Addison-Wesley Educational Publishers Inc; Edición: 2nd edition (16 de noviembre de 2004) 		<ul style="list-style-type: none"> Refactoring. Improving the Design of Existing Code <ul style="list-style-type: none"> Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts Addison Wesley; Edición: 01 (1 de octubre de 1999) 	

Obra, Autor y Edición	Portada	Obra, Autor y Edición	Portada
<ul style="list-style-type: none"> • UML Distilled. A Brief Guide to the Standard Object Modeling Language <ul style="list-style-type: none"> ◦ Martin Fowler, Kendall Scott ◦ Addison-Wesley Educational Publishers Inc; Edición: 3 ed (15 de septiembre de 2003) 		<ul style="list-style-type: none"> • Patrones de diseño <ul style="list-style-type: none"> ◦ Erich Gamma et al ◦ Grupo Anaya Publicaciones Generales; Edición: 1 (1 de noviembre de 2002) 	
<ul style="list-style-type: none"> • Clean Code. A Handbook of Agile Software Craftsmanship <ul style="list-style-type: none"> ◦ Robert C. Martin ◦ Prentice Hall; Edición: 01 (1 de agosto de 2008) 		<ul style="list-style-type: none"> • Object-Oriented Software Construction <ul style="list-style-type: none"> ◦ Bertrand Meyer ◦ Prentice Hall; Edición: 2 ed (3 de abril de 1997) 	

Version 0.0.1

Last updated 2019-09-04 05:59:50 +0200