

Sistemas Complejos

Santa Tecla
parqueNaturalSantaTecla@gmail.com
Version 0.0.1

Índice

Justificación: ¿Por qué?

Necesidades

Universo

Escenarios

Definición: ¿Qué?

Sistema

Sistema Complejo

Objetivos: ¿Para qué?

Efectividad

Descripción: ¿Cómo?

Características de Sistemas Complejos

Capacidades cuantitativas

Capacidades cualitativas

Abstracción

Encapsulación

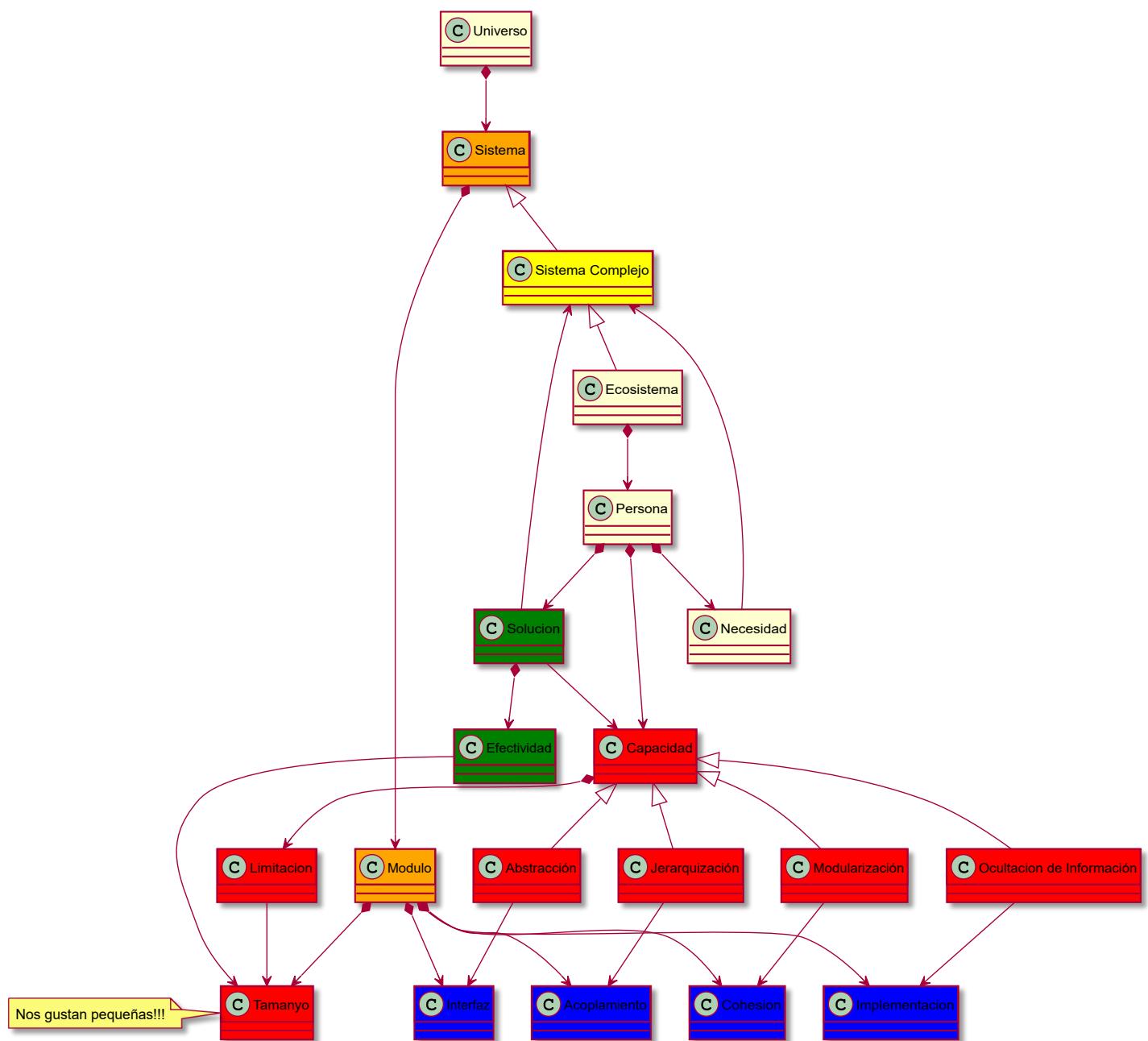
Modularización

Jeararquización

Síntesis

Bibliografía

**Abstracción
Eficiencia
Jerarquización
Modularización
Sistema Eficacia
Capacidad-Cualitativas
Sistema-Complejo
Efectividad
Capacidad-Cuantitativas
Necesidad
Encapsulación
Número-Mágico-de-Miller**



Justificación: ¿Por qué?

Necesidades

“*El hombre es la medida de todas las cosas*

— Protágoras
Grecia Clásica

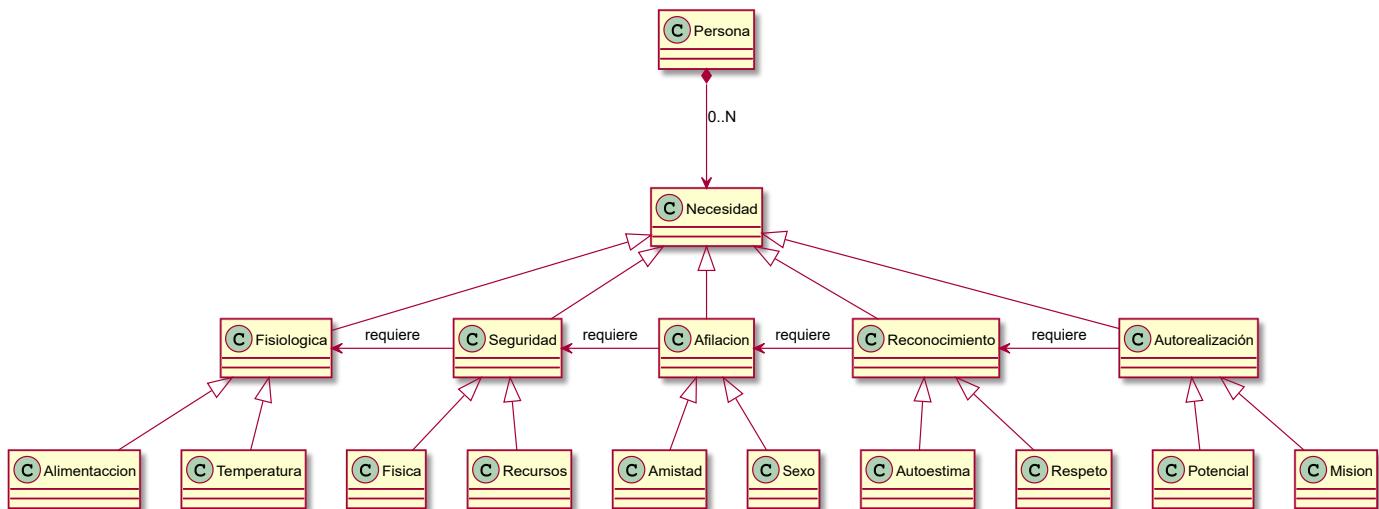
“*EL hombre como centro del universo*

— Humanismo
Renacimiento



Pirámide de Maslow

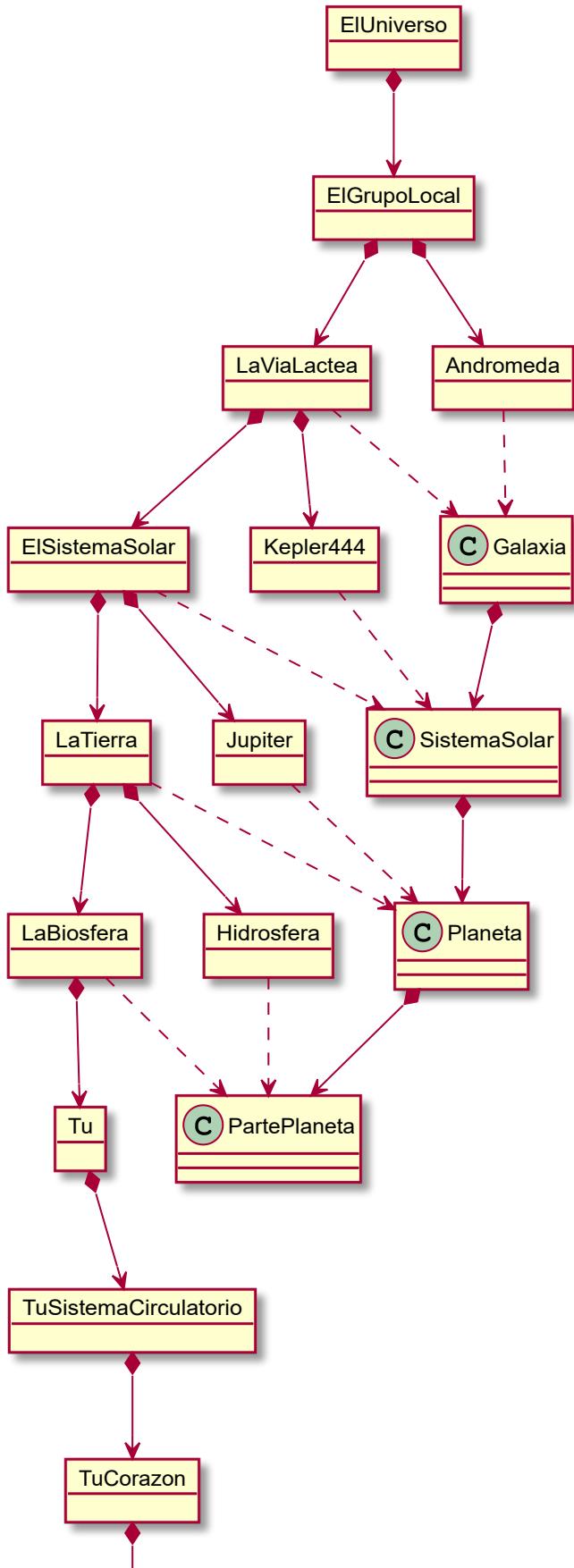
- Para evitar la ambigüedad
 - **UML de RUP:** Lenguaje Unificado de Modelado (*Unified Model Language*) de la metodología del Proceso Unificado de Rational (*Rational Unified Process*)
 - Cada símbolo (léxico) relacionado (sintaxis) en un diagrama tiene un significado estándar (semántica). Así:
 - varias personas entienden lo mismo del mismo diagrama, no como con los gráficos de transparencias, ...
 - tú entiendes hoy el diagrama que dibujaste hace tiempo, no como una servilleta con un garabato, ...

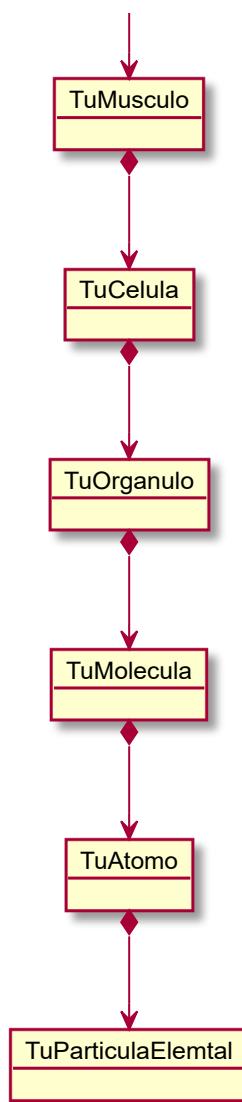


PlantUML	Asciidoc
Generación de diagramas de UML a partir de texto con sintaxis específica	Generación de documentación en HTML, PDF, ... con PlantUML incorporado
www.planttext.com/	asciidoc.org/docs/asciidoc-writers-guide/

Universo

Todas las necesidades se satisfacen con los recursos del universo



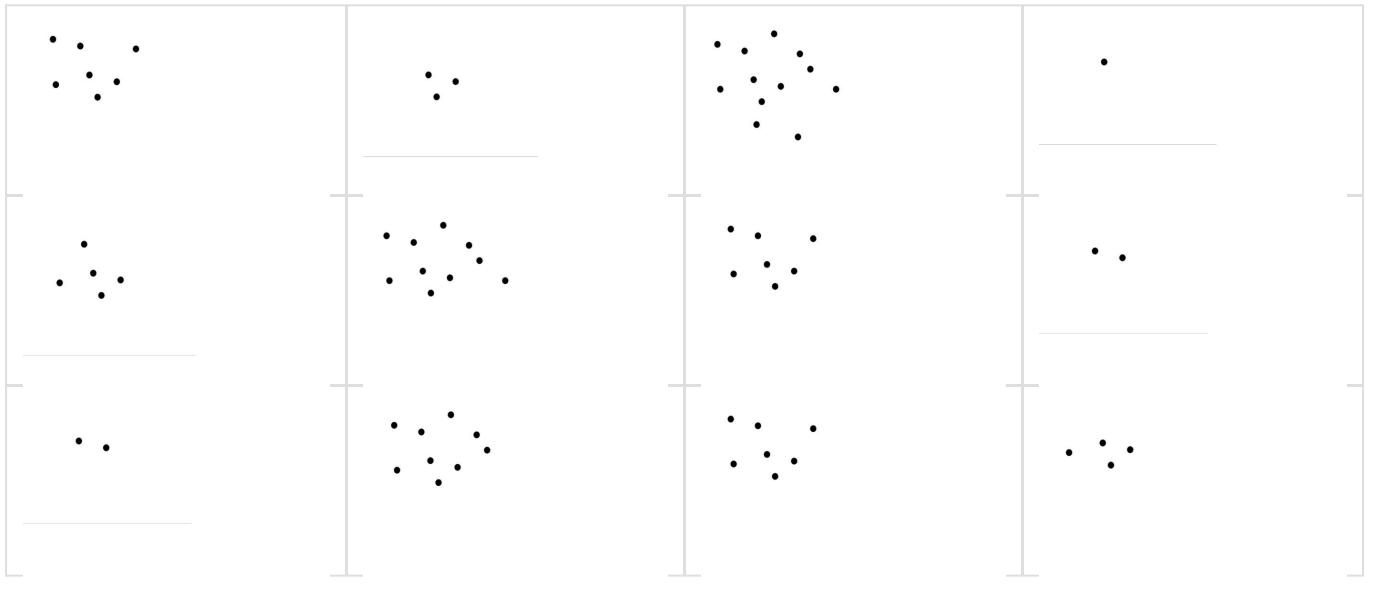


y para explotarlo hay que estudiarlo, conocerlo, ...

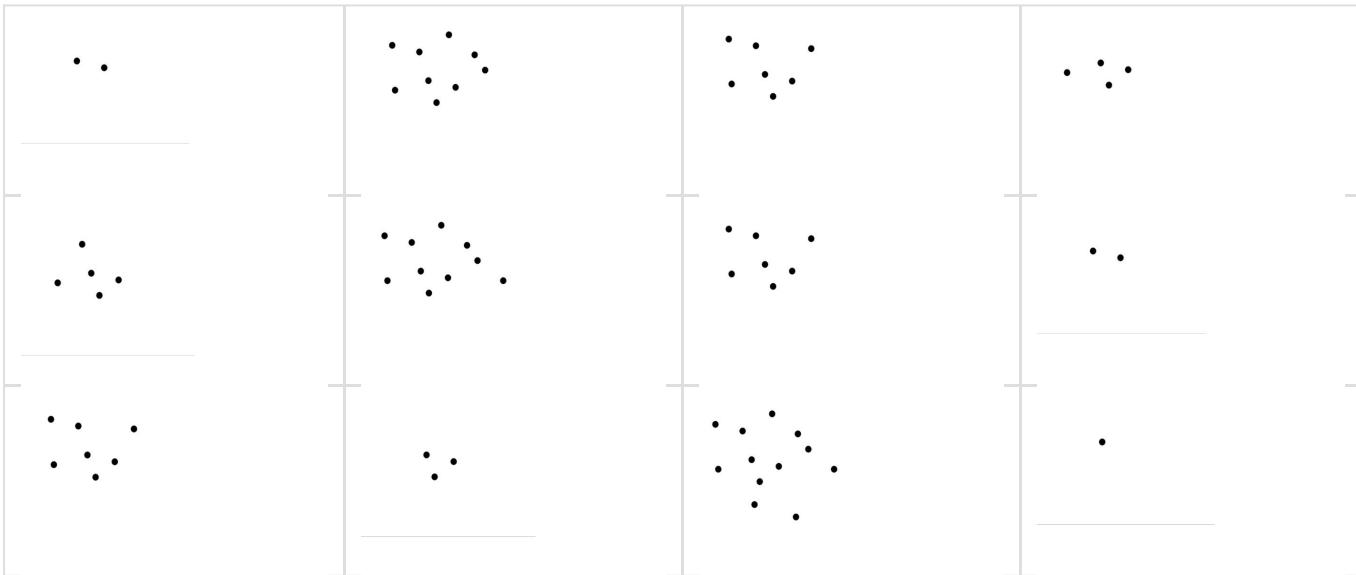
Escenarios

¿Qué tal manejas estos sencillos sistemas? Escenario con Puntos

- Dime cuántos elementos hay en cada imagen



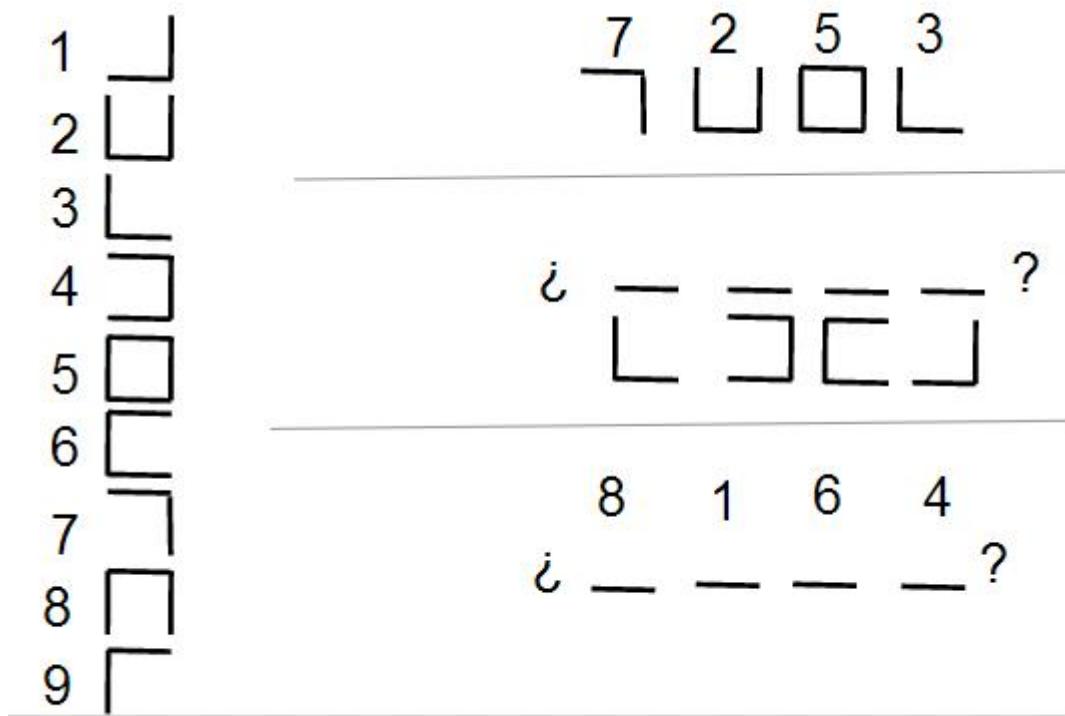
- Dime cuántos elementos hay en cada imagen y sé consciente cuándo los cuentas (siguiendo uno por uno con "golpes" de vista) o cuándo los ves (sin seguimiento)



- La media en las personas es que hasta 4 elementos se ven y a partir de 5 se cuentan

¿Qué tal manejas estos sencillos sistemas? Escenario con Códigos

- Te voy a transmitir una clave con estos códigos, aprendetelos de memoria!



- Te voy a transmitir una clave con estos códigos: cada número con su contorno, aprendetelos de memoria!

7 2 5 3

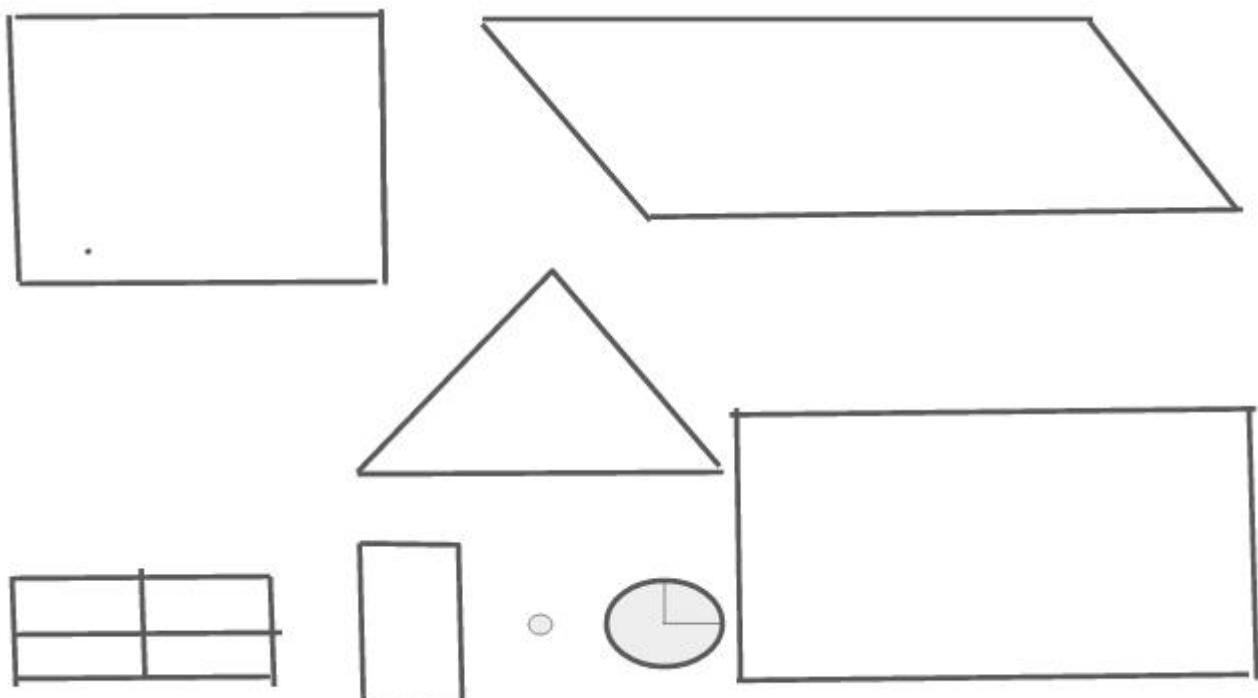
1	2	3
4	5	6
7	8	9

? — □ △ □ ?

8 1 6 4
? — — — ?

¿Qué tal manejas estos sencillos sistemas? Escenario con Figuras

- Mira el dibujo durante 5 segundos para memorizarlo y repetirlo con lápiz y papel

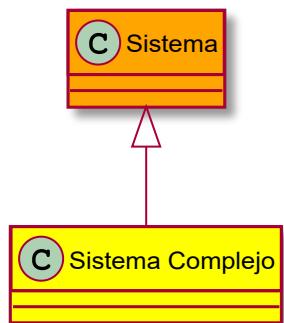


- Mira el dibujo durante 5 segundos para memorizarlo y repetirlo con lápiz y papel



- Asociamos **muchos elementos** en **un todo**

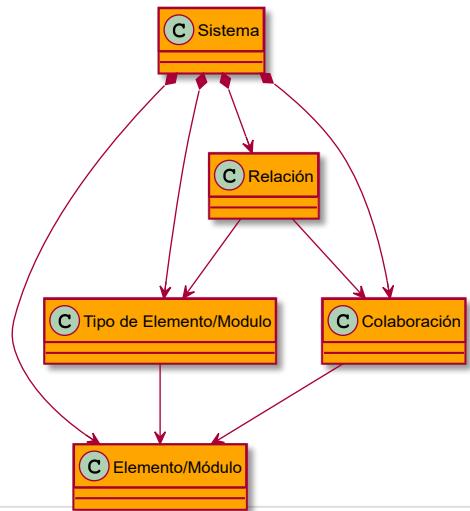
Definición: ¿Qué?



Sistema

“Un Sistema es un conjunto de componentes interactuando o interdependientes formando un todo integrado. Cada sistema está delimitado por sus límites espacio/temporales e influenciado por su entorno, descrito por su estructura y propósito y expresado en su funcionamiento

— Wiki?



Sistema	<i>Sistema respiratorio</i>	<i>Película de amor</i>	<i>Numeros Romanos</i>	<i>Semaforo</i>
conjunto de componentes interactuando o interdependientes formando un todo integrado.	<i>nariz, laringe, faringe, traquea, pulmones, alveolos, ...</i>	<i>personajes</i>	<i>I, V, X, L, C, D y M</i>	<i>Rojo, verde y amarillo</i>
Cada sistema está delimitado por sus límites espacio/temporales	<i>fechas y lugares de la vida del ser vivo que contiene el sistema respiratorio</i>	<i>fechas y lugares de los personajes</i>	<i>fechas y lugar donde estén escritos</i>	<i>fechas y lugar de la instalación del semáforo</i>
e influenciado por su entorno,	<i>lo que respira: aire limpio vs contaminado, ...</i>	<i>la sociedad, las familias, una ex pareja, ...</i>	<i>en desuso en favor de sistemas de numeración posicionales (indo-arábigo, maya, chino, ...) mucho más efectivos</i>	<i>fuente de energía, climatología, vándalos, artistas, ...</i>
descrito por su estructura	<i>la nariz se conecta con la laringe, la laringe con la traquea, ...</i>	<i>argumento que relaciona los personajes de la historia</i>	<i>grupos de máximo 3 elementos consecutivos, grupo con elemento y opcionalmente un elemento inferior prefijo o sufijo, ...</i>	<i>de rojo a verde, de verde a amarillo y de amarillo a verde y/o rojo, ...</i>

Sistema	<i>Sistema respiratorio</i>	<i>Película de amor</i>	<i>Numeros Romanos</i>	<i>Semaforo</i>
y propósito	<i>inyectar oxígeno al sistema circulatorio extrayendo monóxido de carbono, ...</i>	<i>transmitir emociones</i>	<i>registrar información cuantitativa</i>	<i>controlar el tráfico</i>
y expresado en su funcionamiento.	<i>inspiración y expiración</i>	<i>reproducción de la película</i>	<i>suma, resta, producto, división, ... son información cuantitativas registrada</i>	<i>luces con alimentación electrica</i>

Sistema Complejo

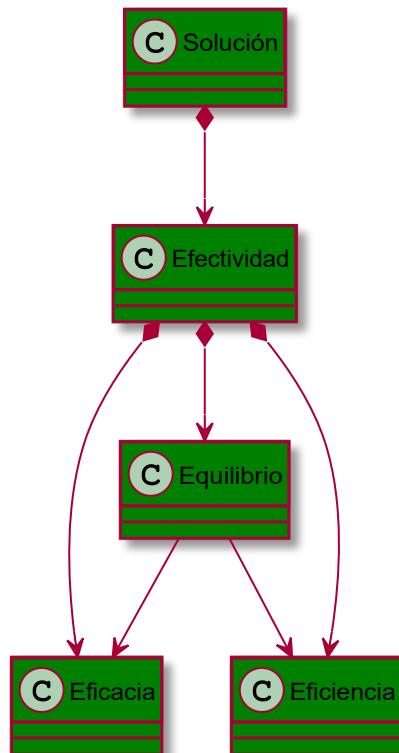
“Un Sistema Complejo es aquel cuya complejidad excede la capacidad intelectual humana”

— Booch

Objetivos: ¿Para qué?

Efectividad

- Efectividad en una capacidad es un **equilibrio** entre:
 - **Eficacia**, alto cumplimiento de objetivos frente a incumplimientos por errores, desmotivación, cansancio, ...
 - **Eficiencia**, bajo consumo de recursos (tiempo, energía, espacio, ...) frente a consumos disparatados comparado con otras soluciones

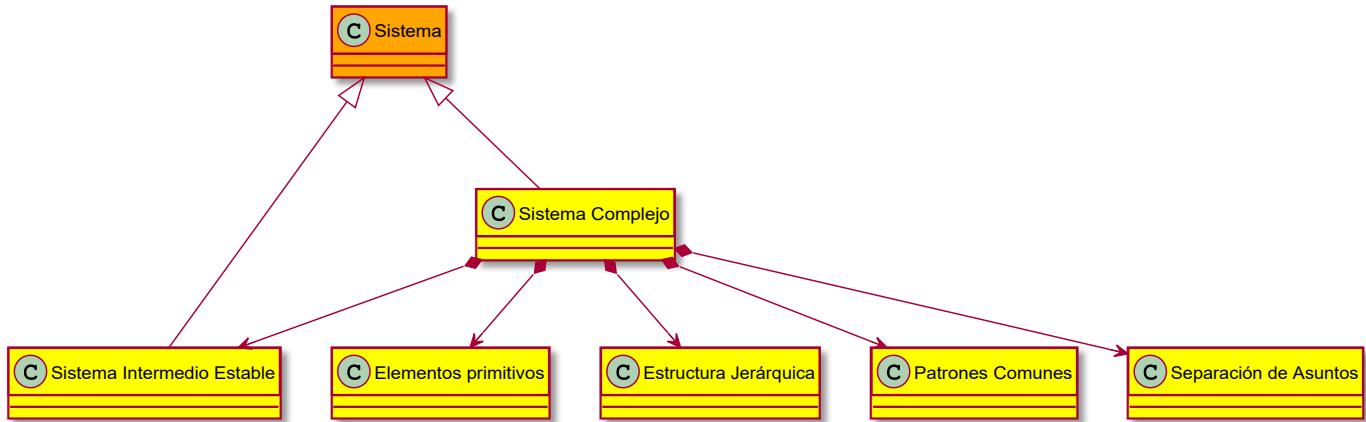


EFICACIA +	(A) EFICAZ e INEFICIENTE Haber alcanzado los retos sin cumplir con las pautas	(B) EFICAZ y EFICIENTE Haber alcanzado los retos con los recursos dispuestos
	(C) INEFICAZ e INEFICIENTE Haber fracasado en el cumplimiento de objetivos pese a extralimitar el uso de los medios	(D) INEFICAZ y EFICIENTE Haber utilizado bien los recursos disponibles sin alcanzar retos
EFFECTIVIDAD	-	EFICIENCIA

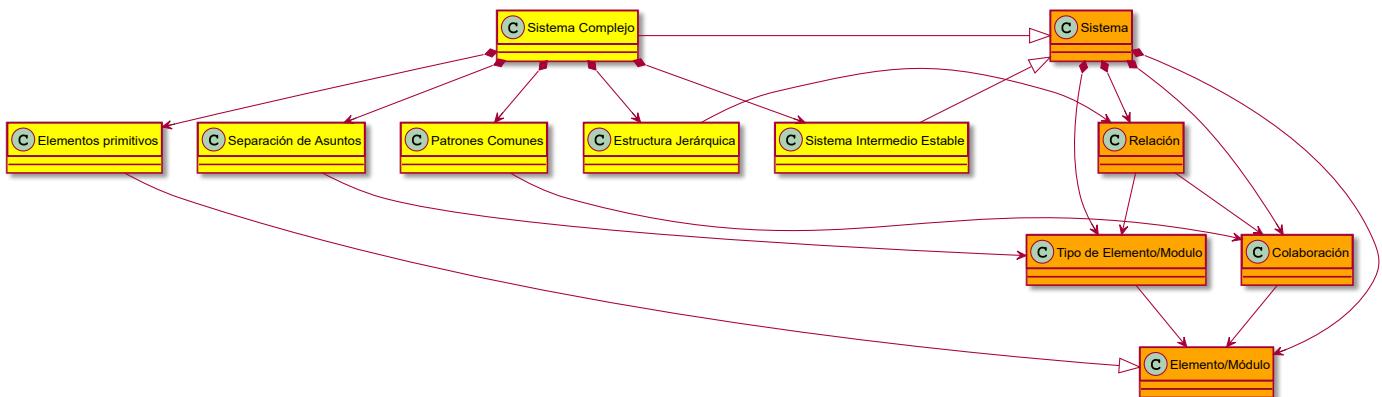
Efectividad para Comprar el pan	Eficientemente	Ineficientemente
<i>Eficazmente</i>	<i>Trae el pan en cinco minutos y devuelve la vuelta</i>	<i>Trae el pan en 20 minutos y no devuelve la vuelta</i>
<i>Ineficazmente</i>	<i>Trae el pan pero no la cantidad, tipos, ... oportunos en cinco minutos y devuelve la vuelta</i>	<i>Trae el pan pero no la cantidad, tipos, ... oportunos en 20 minutos y no devuelve la vuelta</i>

Descripción: ¿Cómo?

Características de Sistemas Complejos

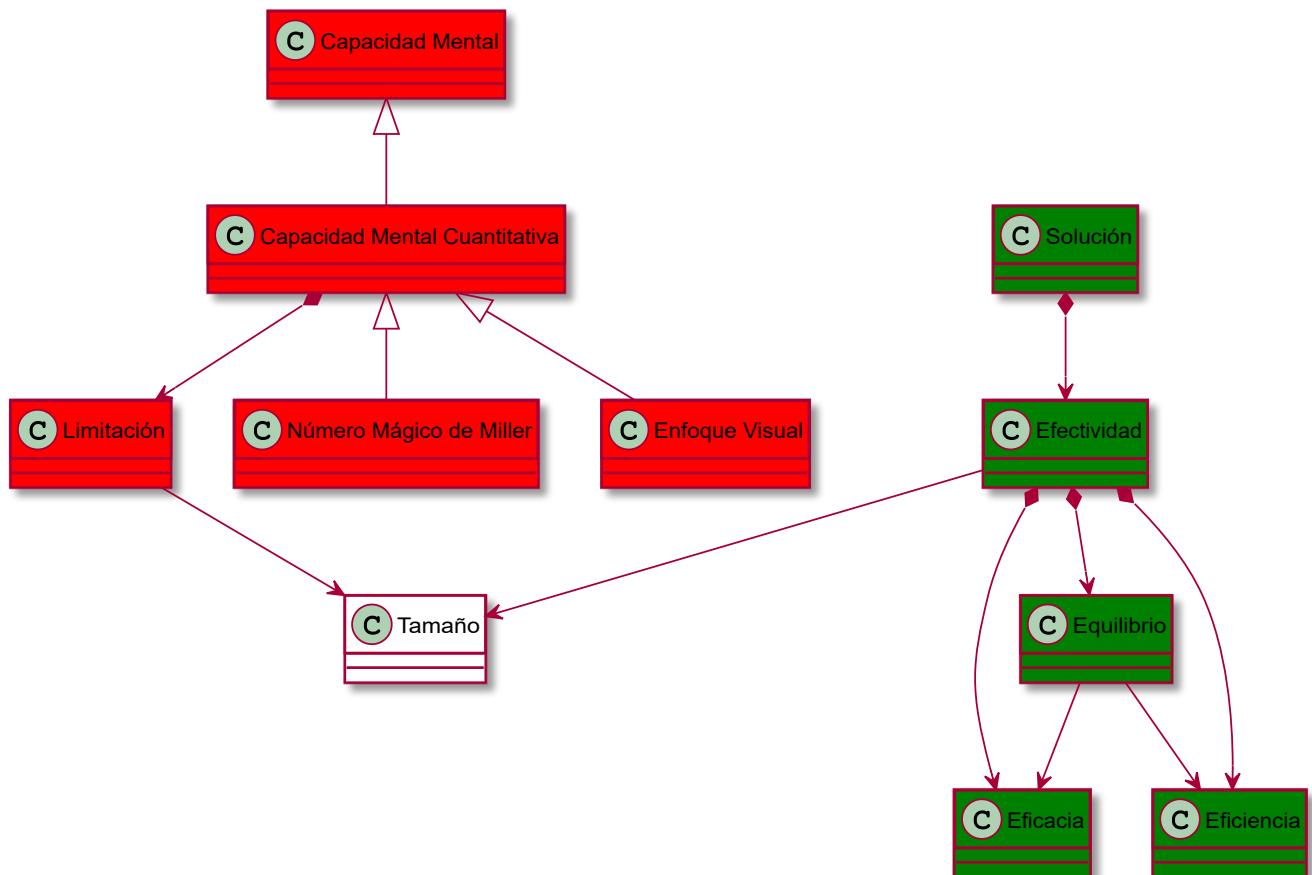


- **Estructura jerárquica.** Frecuentemente, la complejidad adquiere una forma jerárquica donde el sistema complejo está compuesto de subsistemas interrelacionados que a su vez tienen sus propios subsistemas y así hasta que se alcanza algún elemento del más bajo nivel. No solo son sistemas complejos jerárquicos sino que los niveles de su jerarquía representan los diferentes niveles de abstracción cada uno construido sobre otro y cada uno comprensible por sí mismo. En cada nivel de abstracción, encontramos una colección de elementos que colaboran para proveer servicios a niveles superiores
- **Elementos primitivos relativos.** La elección de qué componentes en un sistema son primitivos es relativamente arbitraria y mayormente está a discreción del observador del sistema
- **Separación de asuntos.** Las intra-conexiones de componentes son más fuertes que las inter-conexiones de componentes. Este hecho tiene el efecto de separar los componentes con dinámica de alta frecuencia (involucrando la interacción entre componentes) de los de dinámica de baja frecuencia. En términos sencillos, hay una clara separación de asuntos entre las partes de diferentes niveles de abstracción
- **Patrones comunes.** Los sistemas jerárquicos se componen generalmente de sólo unos pocos tipos diferentes de subsistemas en varias combinaciones y órdenes. Nos encontramos con una gran similitud en la forma de mecanismos compartidos unificando esta vasta jerarquía
- **Formas intermedias estables.** Un sistema complejo que funciona invariablemente se encuentra que ha evolucionado a partir de un sistema sencillo que funcionó. Un sistema complejo diseñado desde cero no funciona y no puede ser remendado para hacer que funcione. Usted tiene que comenzar de nuevo, a partir de un sistema sencillo de trabajo



Capacidades cuantitativas

- Con **Limitación para el Tamaño** de la carga del área de trabajo
- Inefectividad
- **Ineficiencia**
 - **Ley de Hyks:** tiempo que tarda una persona en tomar una decisión respecto a la cantidad de posibles elecciones que tenga aumenta el tiempo de decisión logarítmicamente según aumenta el número de opciones
- **Ineficacia**
 - **Número Mágico de Miller:** el número de objetos que una persona promedio puede tener en la memoria de trabajo es 7 ± 2 . *Ejemplos:*
 - Película de espías, 21 gramos, ... muchas historias paralelas poco relacionadas hasta el final
 - Código de números en cuadrícula, ... muchas frente a pocas reglas
 - Lenguaje natural, ... muchas palabras en muchas estructuras sintácticas
 - **Enfoque Visual:** Ver hasta 4 elementos vs Contar a partir de 5 elementos
 - Incluso no ves lo que ves (*contar pases de pelota con ...*)



Ejemplo: Personas

- Las personas no son efectivas con tamaños elevados de elementos a gestionar por imposibilidad, agotamiento, desmotivación, distracción, ...)
 - No son eficaces por errores (¿nunca te has equivocado?, ...)
 - No son eficientes por
 - consumo de tiempo (calcula los primos del primer billón, a relación de un segundo por cálculo de si un número es primo, no hay segundos en la vida de una persona, ...)
 - consumo de espacio (calcula con lapiz y papel los posibles caminos entre todas las poblaciones de tu tierra natal, suponiendo que habrá miles, no hay hojas en el planeta, ...)

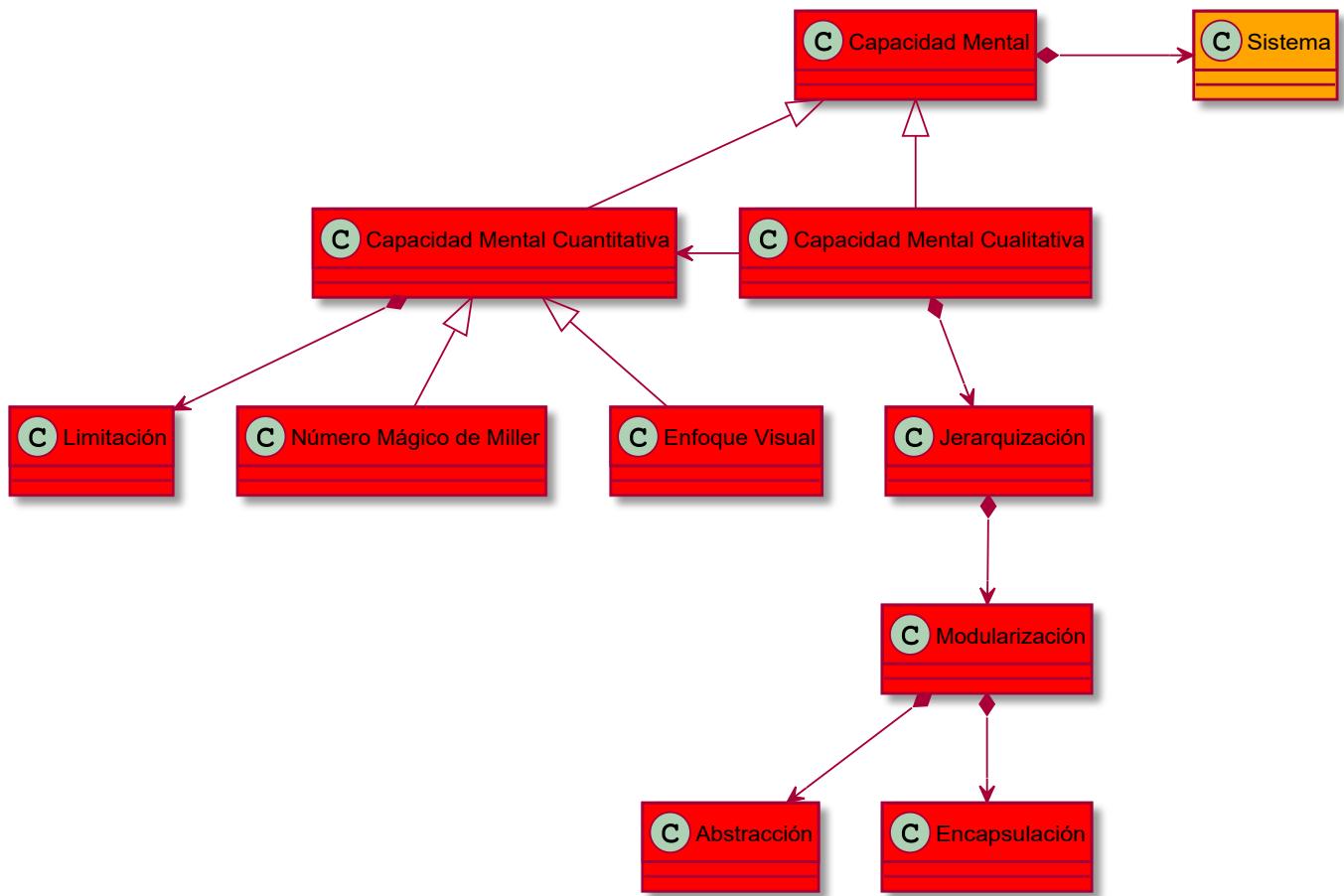
“*Divide et impera (divide y vencerás)*

— *Imperio Romano*

Julio César

Capacidades cualitativas

- La historia del ser humano disfruta de cuatro mecanismos mentales que facilitan enormemente nuestra comprensión de los sistemas complejos:
 - **Abstracción**
 - **Encapsulación**
 - **Modularización**
 - **Jerarquización**



Abstracción

“La abstracción surge de un reconocimiento de similitudes entre ciertos objetos, situaciones o procesos en el mundo real, y la decisión de concentrarse en estas similitudes e ignorar por el momento, las diferencias

— Dahl
Dijkstra y Hoare

“La abstracción es "una descripción simplificada, o especificación, de un sistema que hace hincapié en algunos de los detalles o propiedades mientras que suprime otros del sistema. Una buena abstracción es la que hace hincapié en los detalles que son importantes para el lector o usuario y suprime detalles que son, al menos por ahora, de distracción

— Shaw

“La abstracción es el proceso mental de extracción de las características esenciales de algo, ignorando los detalles superfluos

— Booch

- Implicaciones:

- Una abstracción denota las **características esenciales** de un objeto que lo distinguen de todos los otros tipos de objetos y por lo tanto proporciona **límites conceptuales nítidamente definidos**, en relación con la perspectiva del espectador.
- Una abstracción se centra en la **visión exterior** de un objeto y sirve para separar el comportamiento esencial de un objeto de su implantación
- La abstracción es **eminentemente subjetiva**, dependiendo del interés del observador
- Nos esforzamos para construir abstracciones de las **entidades porque son directamente paralelos al vocabulario del dominio de un determinado problema**.

Ejemplo: Varios

- Mundo real: un autobús de un pasajero o un mecánico, un ordenador, ...
- Software orientado a procesos: factorial, mostrar menú, ordenar, ...
- Software orientado a objetos: una fecha, un intervalo, un gestor de comunicaciones, un colección de datos, ...

Encapsulación

“La encapsulación es proceso por el que se ocultan los detalles del soporte de las características esenciales de una abstracción

— Booch

- Hacer notar que en ninguno de los casos **no se trata de ocultar la información** en sí misma sino de ocultar los detalles del soporte de dicha información
- La encapsulación se logra con mayor frecuencia a través de ocultación de información, que es el proceso de **ocultar todos los secretos de un objeto que no contribuyen a sus características esenciales**

- La encapsulación proporciona barreras explícitas entre las diferentes abstracciones y por lo tanto conduce a una clara **separación de asuntos**. El beneficio inmediato será la posibilidad de cambiar los soportes de las características de una abstracción sin afectar a todos los elementos que dependan de esas características porque ni los conocen, ni los mencionan
 - Principio de Encapsulación: **todo aquello que no sea necesario dar a conocer, no se debe dar a conocer**
- Implicaciones:
 - Una vez realizada cierta abstracción hay que “trasladarlas” al lenguaje de programación. Esto conlleva decidir entre diversas estructuras de datos (estáticas o dinámicas, en memoria principal o secundaria, etc.) y/o diversos algoritmos (¿con variables auxiliares o no? ¿recursivo o iterativo?, etc.), siendo diversas las alternativas que recogen dichas características esenciales. Una vez que se selecciona una implantación, debe ser tratado como un secreto de la abstracción y oculta a la mayoría de los clientes. En la práctica, esto significa que cada clase debe tener dos partes:
 - La **interfaz** de una clase capta sólo su vista exterior, que abarca nuestra abstracción del comportamiento común a todas las instancias de la clase. La interfaz de una clase es el único lugar donde establecemos todas las suposiciones que un cliente puede hacer sobre cualquier instancia de la clase
 - La **implementación** de una clase comprende la representación de la abstracción, así como los mecanismos para conseguir el comportamiento deseado. La implementación encapsula detalles sobre los qué ningún cliente puede hacer suposiciones.
 - La **abstracción de un objeto debe preceder a las decisiones acerca de su implantación.**
 - **Ninguna parte de un sistema complejo debe depender de los detalles internos de cualquier otra parte.** Mientras que la abstracción ayuda a las personas a pensar en lo que están haciendo, la encapsulación permite hacer cambios fiables en el programa con un esfuerzo limitado.

Ejemplo: Varios

- Mundo real: un autobús, un ordenador, una universidad, ...
- Software orientado a procesos: factorial, mostrar menú, ordenar, ...
- Software orientado a objetos: una fecha, un intervalo, un gestor de comunicaciones, un colección de datos, ...

Modularización

“La modularidad es el proceso de descomposición de un sistema en un conjunto de piezas poco acoplados y cohesivos

— Booch
96

“El acoplamiento” [...] es la medida de fuerza de la asociación establecida por una conexión entre un módulo -elemento- y otro. El acoplamiento fuerte complica un sistema porque los módulos son más difíciles de comprender, cambiar o corregir por sí mismos si están muy interrelacionados con otros módulos

— Booch
96

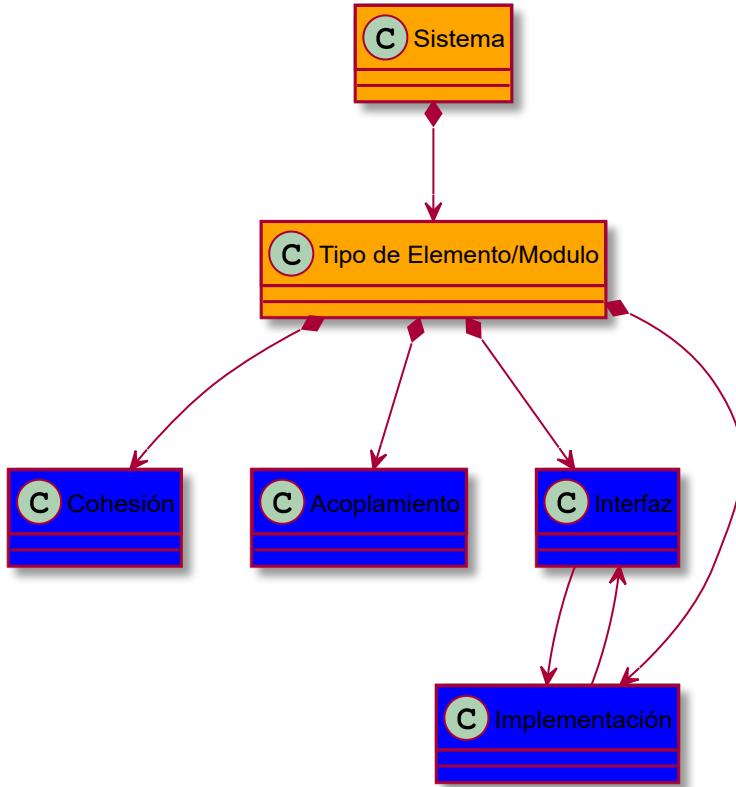
“La cohesión mide el grado de conectividad entre los elementos de un solo módulo

— Booch
96

- Al diseñar un sistema de software complejo, es esencial para descomponer en partes más pequeñas y más pequeñas, cada una de las cuales podemos entonces refinar independientemente. De esta manera, satisfacemos la restricción muy real que existe sobre **la capacidad del canal de la cognición humana**: para entender cualquier nivel dado de un sistema, sólo necesitamos comprender algunas partes (en lugar de todas las partes) a la vez.
 - Para modularizar hay que **minimizar las dependencias entre módulos** (acoplamiento) que deben tener significado propio por sí mismo **agrupando abstracciones lógicamente relacionadas** (cohesión)
 - El **bajo acoplamiento de un modulo se basa en la abstracción** que limita su interfaz a lo esencial y **en la encapsulación** que oculta todos los detalles necesarios para su implantación pero innecesarios para otros módulos que se relacionen con éste
 - Dividir un programa en una serie de límites documentados bien definidos dentro del programa es de **gran valor en la comprensión del programa**
 - Debería ser posible cambiar la implementación de otros módulos sin el conocimiento de la aplicación de otros módulos y sin afectar el comportamiento de los otros módulos

“La descomposición inteligente se dirige directamente a la complejidad inherente del software al obligar a una división del espacio de estados de un sistema

— Parnas



Ejemplo: Varios

- Mundo real: un autobús, un ordenador, una universidad, ...
- Software orientado a procesos: factorial, mostrar menú, ordenar, ...
- Software orientado a objetos: una fecha, un intervalo, un gestor de comunicaciones, un colección de datos, ...

Jerarquización

“Jerarquía es una clasificación u ordenamiento de las abstracciones”

— Booch

“La jerarquización es el proceso de estructuración por el que se produce una organización de un conjunto de elementos en grados o niveles de responsabilidad, de clasificación o de composición, ... entre otros”

— Fernando Arroyo

- Implicaciones:
 - La abstracción es una buena cosa pero en todos los casos, excepto las aplicaciones más triviales, podemos encontrar muchas más abstracciones diferentes de lo que podemos comprender a la vez. La encapsulación ayuda a gestionar esta complejidad al ocultar el interior de la vista de nuestras abstracciones. La modularidad ayuda también, por que nos da una manera de agrupar abstracciones relacionados lógicamente. Aún así, esto no es suficiente. Un conjunto de abstracciones a menudo forma una jerarquía, y mediante la identificación de estas jerarquías en nuestro diseño se simplifica enormemente nuestra comprensión del problema.

- La identificación de las jerarquías dentro de un sistema de software complejo a menudo **no es fácil**. Una vez que se exponen esas jerarquías, la estructura de un sistema complejo se vuelve muy simple y obtenemos la comprensión de la misma.
- Si no revelamos la estructura de clases de un sistema, tendríamos que multiplicar nuestro conocimiento sobre las propiedades de cada parte individual. Con la inclusión de la estructura de clases, captamos estas **propiedades comunes en un solo lugar**.
- Existen **dos jerarquías ortogonales del sistema: la estructura de clases y la estructura de objetos**. Cada jerarquía está en capas, con clases más abstractas y objetos construidos sobre otros más primitivos. La clase u objeto que se elija como primitivo está en relación con el problema en cuestión. Mirando dentro de cualquier nivel dado revela otro nivel de complejidad. Especialmente entre las partes de la estructura del objeto, existe una estrecha colaboración entre los objetos de ese mismo nivel de abstracción.
- **La estructura de clases y la estructura de objetos no son completamente independientes**; más bien, cada objeto en la estructura de objetos representa una instancia específica de una clase. Por lo general hay muchos más objetos que clases de objetos dentro de un sistema complejo. Al mostrar la "parte de", así como la jerarquía "es un", exponemos de forma explícita la redundancia del sistema considerado.
- La mayoría de los sistemas interesantes **no incorporan una única jerarquía**; en cambio, nos encontramos con que muchas jerarquías diferentes suelen estar presentes dentro del mismo sistema complejo. En nuestra experiencia, hemos encontrado que es esencial para ver un sistema desde ambas perspectivas, estudiando su jerarquía "es un" (clasificación), así como su jerarquía "parte de" (composición)

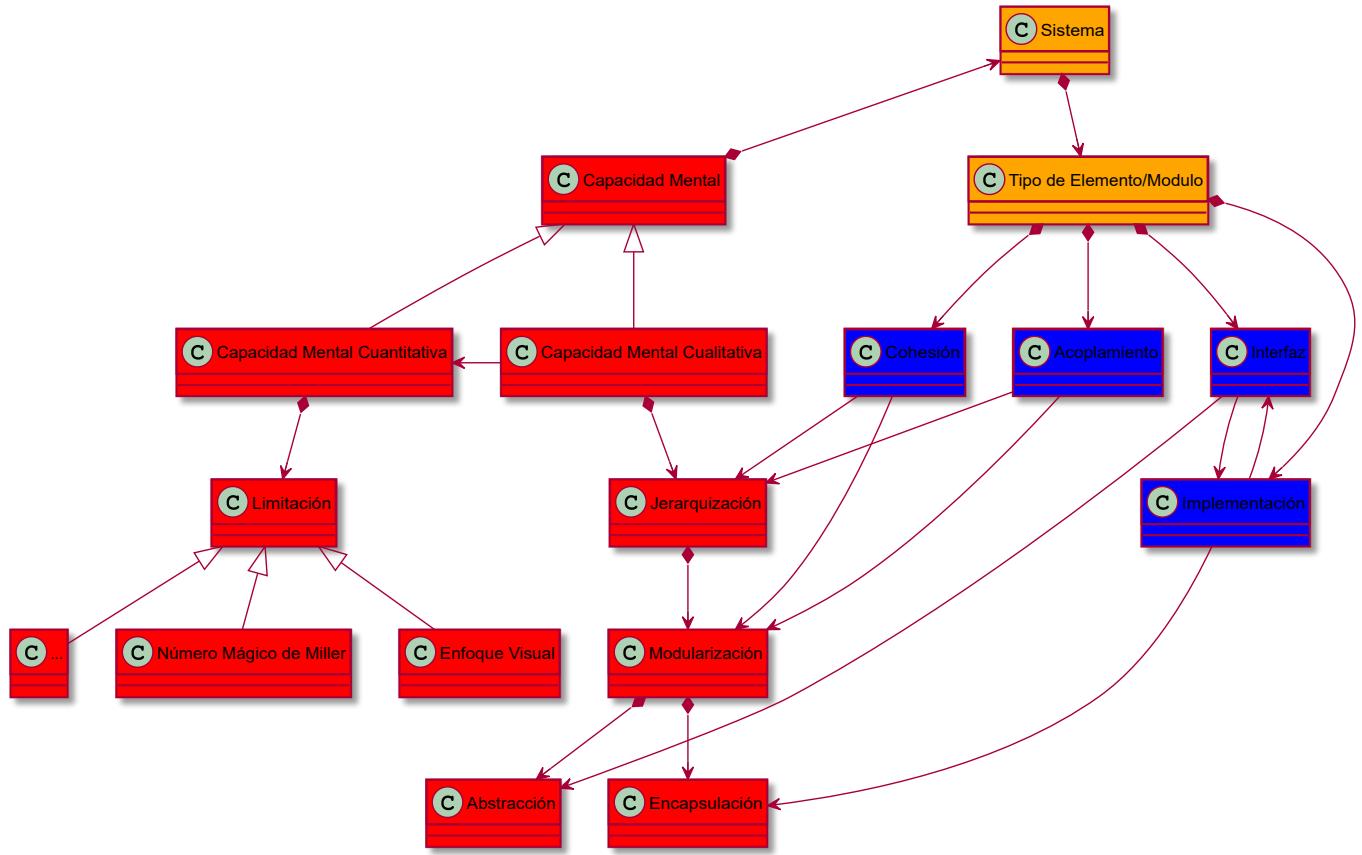
“Nuestra experiencia es que los sistemas de software complejos más exitosos son aquellos cuyos diseños incluyen explícitamente las estructuras de clases y objetos bien diseñadas y encarnan los cinco atributos de sistemas complejos descritos en la sección anterior. [...] Muy raramente nos encontramos con sistemas de software que se entregan a tiempo, que están dentro del presupuesto y que cumplen con sus requisitos, a menos que estén diseñados con estos factores en mente

— Booch

Ejemplo: Varios

- Mundo real: un autobús, un ordenador, una universidad, ... se componen de otros elementos y los hay de varias clases similares
- Software orientado a procesos: factorial, mostrar menú, ordenar, ... se componen de otros elementos y los hay de varias clases similares
- Software orientado a objetos: una fecha, un intervalo, un gestor de comunicaciones, un colección de datos, ... se componen de otros elementos y los hay de varias clases similares

Síntesis



Conocimiento

Santa Tecla
parqueNaturalSantaTecla@gmail.com
Version 0.0.1

Índice

Justificación: ¿Por qué?

Definición: ¿Qué?

Pirámide DIKW

Objetivos: ¿Para qué?

Descripción: ¿Cómo?

Historia de la Ciencia

Método Científico de Galileo

Método Cartesiano

Ciclo de Deming

Dialéctica Hegeliana

Paradigmas de Khun

Conjuntos

Tipos de Conjuntos

Tipos de Elementos

Recursividad

Jerarquías de Composición vs Clasificación

Asociación

Estructuras

Estrategias de Clasificación

Categorización clásica

Agrupación conceptual

Teoría de Prototipos

Bibliografía

Justificación: ¿Por qué?

“*Conocer para dominar la naturaleza*

— Humanismo
Renacimiento

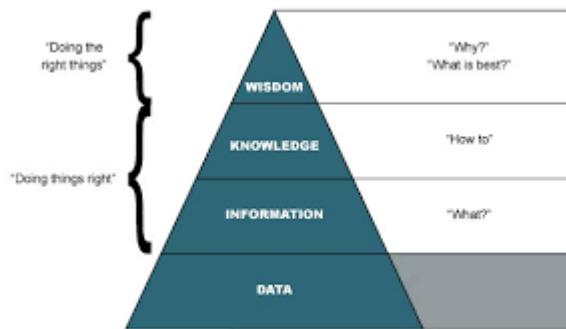
- *Ejemplos:*

- *Encontrar el camino más corto para visitar diferentes puntos*
- *Donde ubicar un aeropuerto que minimice el acceso desde tres poblaciones*
- *Cuántos focos de iluminación colocar para iluminar un sala*
- *Posología de un medicamento, qué, cuándo, dónde y cómo, para curarme*
- ...

Definición: ¿Qué?

Pirámide DIKW

- Pirámide de Datos/Información/Conocimiento/Sabiduría
 - **DIKW: Data Information Knowledge Wisdom**



Nivel	Definición	Ejemplos
Ruido	Señales físicas (visuales, sonoras, ...) no estandarizadas, sin formar parte de un código	ñlaksjdfk, brrrrrr, xvi, ...
Datos	son un hecho concreto o cifras sin ningún contexto o carentes de significado. Sin nada más que los definen, estos dos elementos de datos no tienen mucho sentido.	1650, 9,6%, \$709.7 miles de millones
Información	Es la aplicación de un orden estructurado a los datos con el propósito de que tengan algún significado. La información es un dato que está organizado.	1650 son los puntos de S&P, 9,6% es la desocupación y \$709.7 miles de millones fue el PIB de Argentina en el 2011.
Conocimiento	La comprensión de un tema específico, a través de la experiencia (o educación). Normalmente se utiliza el conocimiento en términos de una habilidad o pericia personal en un área determinada. El conocimiento general refleja una comprensión empírica, más que intuitiva. Se construye por sobre la información para darnos un contexto. La diferencia clave entre el conocimiento y la información es que el conocimiento nos da poder para tomar medidas.	Te desenvuelves en un tipo de fiesta porque la conoces
Sabiduría	Es el juicio óptimo , lo que refleja un profundo conocimiento de las personas, cosas, eventos o situaciones. Una persona que tiene la sabiduría puede aplicar efectivamente la percepción y el conocimiento con el fin de producir los resultados deseados. Es la comprensión de la realidad objetiva dentro de un contexto más amplio.	Sabe divertirse en una fiesta

Objetivos: ¿Para qué?

- Para obtener **soluciones efectivas**, eficaces y eficientes, a partir de **sistemas complejos** mediante:

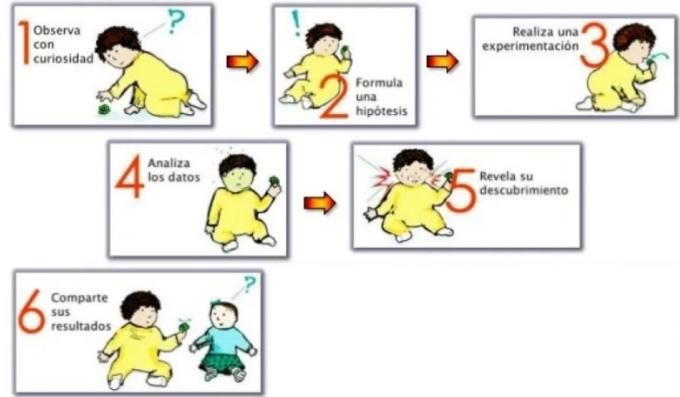
Sector	Descripción	Actividades
Primario	Conformado por las actividades económicas relacionadas con la transformación de los recursos naturales en productos primarios no elaborados . Usualmente, los productos primarios son utilizados como materia prima en otros procesos de producción en las producciones industriales.	Las principales actividades del sector primario son la agricultura , la ganadería , la silvicultura , la apicultura , la acuicultura , la caza , la pesca y piscicultura y la minería , aunque algunos consideran a la minería parte del sector industrial de las regiones.
Secundario	Reúne la actividad artesanal e industrial manufacturera, mediante las cuales los bienes provenientes del sector primario son transformados en nuevos productos.	Este sector se divide en dos sub-sectores: el industrial extractivo, que son la industria minera y petrolífera , y el industrial de transformación como las actividades de envasado , embotellado , manipulación y la transformación de materias primas y/o productos semi-elaborados .
Terciario	Es un sector que no produce bienes , pero que es fundamental en una sociedad capitalista desarrollada. Su labor consiste en proporcionar a la población todos los productos que fabrica la industria, obtiene la agricultura e incluso el propio sector servicios	Ofrecer servicios a la sociedad, a las personas y a las empresas, lo cual significa una gama muy amplia de actividades que está en constante aumento. Ésta abarca desde el comercio más pequeño , hasta las altas finanzas o el Estado.
Cuaternario o de información	Actividades relacionadas con el valor intangible de la información , abarcando la gestión y la distribución de dicha información. Este nuevo enfoque surge del concepto de sociedad de la información o sociedad del conocimiento , cuyos antecedentes se remontan al concepto de sociedad postindustrial, acuñado por <i>Daniel Bell</i> .	Dentro de este sector se engloban actividades especializadas de investigación , desarrollo , innovación y divulgación (I+D+i+d)
Quinario	Relativo a los servicios sin ánimo de lucro	Relacionados con la cultura , la educación , el arte y el entretenimiento . Sin embargo, las actividades incluidas en este sector varían de unos utores a otros, incluyendo en ocasiones actividades relacionadas con la sanidad.

Descripción: ¿Cómo?

Historia de la Ciencia

Método Científico de Galileo

- Método científico:
 - la **observación** del sistema en el **pasado**
 - la **hipótesis** del sistema
 - la **experimentación** con el sistema en el **presente**
 - la **predicción** del sistema en el **futuro** con **patrones, leyes, principios, teoremas, heurísicas, ...**
 - la **publicación** de resultados para **verificación y reusabilidad** por parte de otros

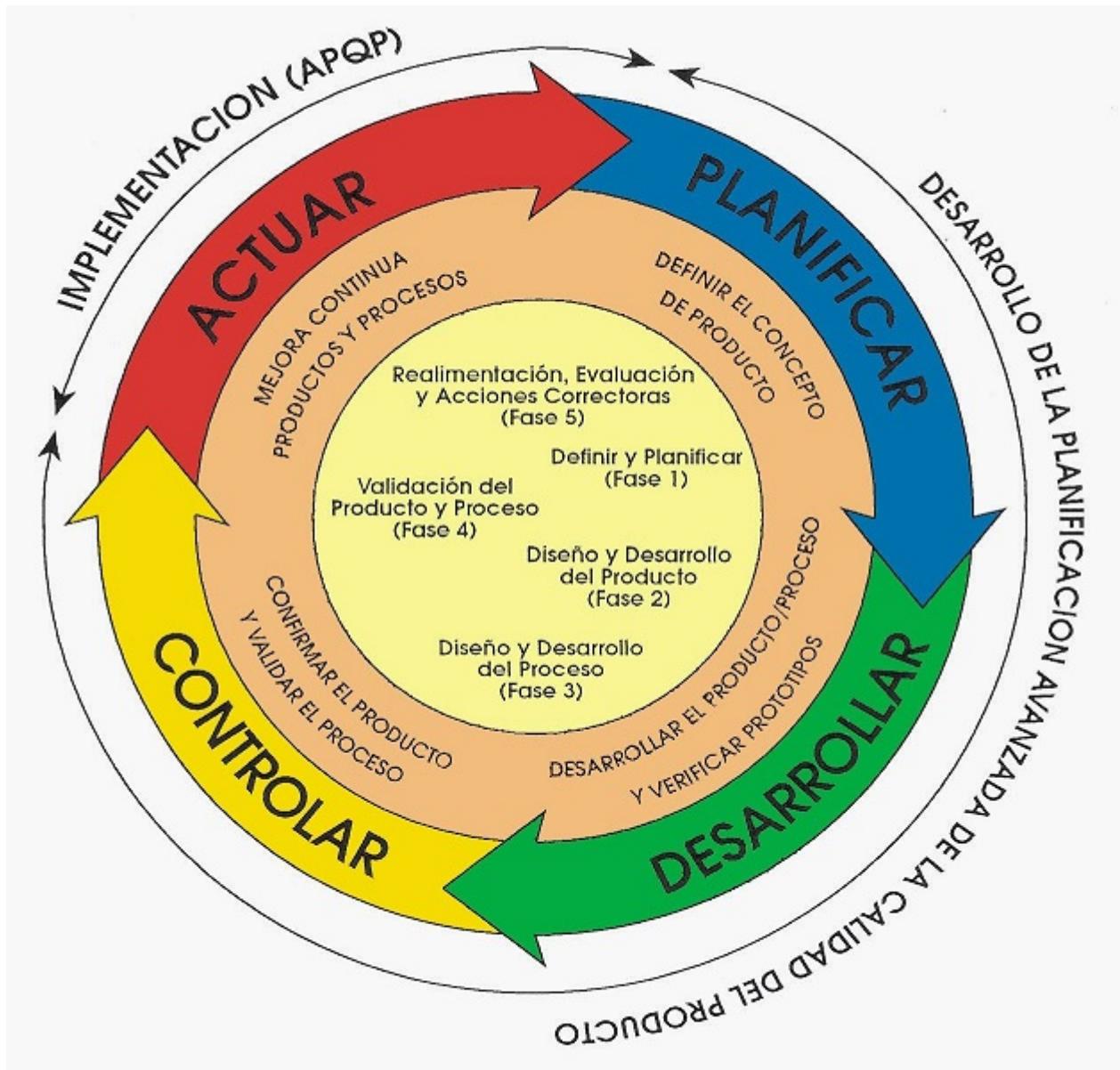


Método Cartesiano

- **Principio de duda o evidencia sistemática**, no aceptar como verdadero algo hasta que se compruebe con evidencia, clara y distintamente, aquello que es realmente verdadero. Con la duda sistemática, se evita la prevención y la precipitación, aceptándose como cierto lo que sea evidentemente cierto.
- **Principio del análisis o descomposición**, dividir y descomponer cada dificultad o problema en **tantas partes como sea posible y necesarias para su comprensión** y solución y resolverlas por separado.
- **Principio de la síntesis o la composición**, conducir cuidadosamente los pensamientos y razonamientos, a partir de las formas más fáciles y simples de conocer para pasar gradualmente a los más difíciles, y así ir armando pensamientos para poder probar su funcionamiento.
- **Principio de la enumeración o de la verificación**, hacer verificaciones, recuentos y revisiones para asegurarse de que nada fue omitido o pasado por alto, y poder comprobar si tu evidencia es falsa o verdadera

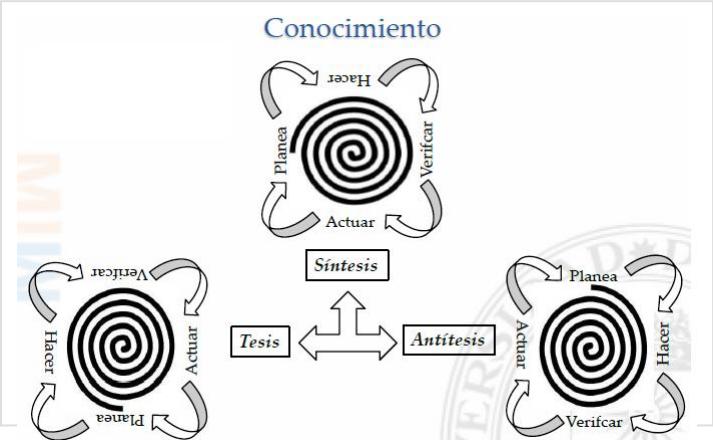
Ciclo de Deming

- Estrategia de mejora continua de la calidad



Dialéctica Hegeliana

La dialéctica se basa en la fundamentación de que una idea (**tesis**), generalmente histórica, social o filosófica, al ser desarrollada en detalle, abre aspectos diversos que entre sí se avienen mal (**antítesis**), pero finalmente surge una manera de reconcebirla conciliando aspectos aparentemente contradictorios (**síntesis**)



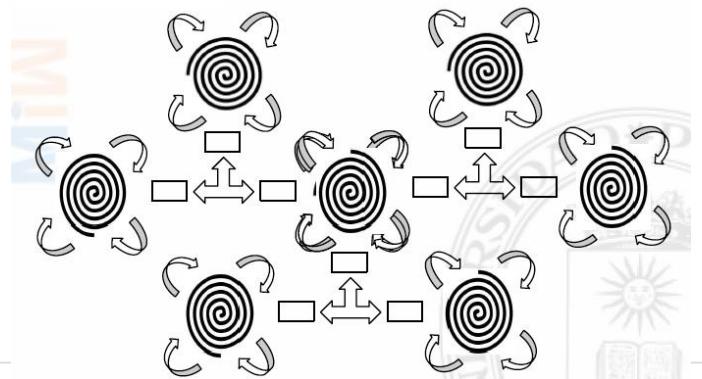
Paradigmas de Khun

“Paradigma: conjunto de prácticas y saberes que definen una disciplina científica durante un período específico

—Kuhn

Estructura de las Revoluciones Científicas

Conocimiento

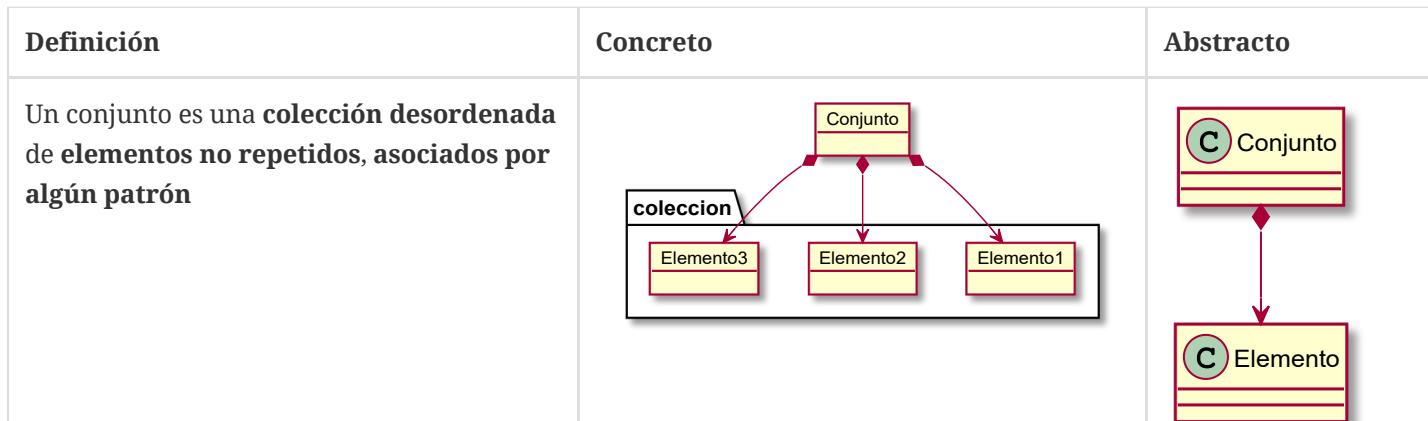


- **Ejemplos:**

- **Física:** la física de Newton (tesis, tiempo y espacio absolutos) no explicaba sucesos en áreas específicas (antítesis, no se ve a Mercurio donde debería estar, ...), hasta que la física de Einstein (síntesis, tiempo-espacio relativo) da una explicación general
- **Filosofía:** los racionalistas (tesis, acentúa el papel de la razón en la adquisición del conocimiento de Platón, Descartes, ...) frente a los empiristas (antítesis, todo conocimiento deriva de la experiencia sensible, ésta es la única fuente de conocimiento de Aristóteles, Hume, ...), hasta el criticismo (síntesis, con juicios a priori de Kant)
- **Hardware:** procesadores CISC (tesis, complejas instrucciones potentes), procesadores RISC (antítesis, pequeñas instrucciones paralelizables), hasta ...

Conjuntos

- **Fundamento** del conocimiento, del pensamiento, del lenguaje y de la lógica: cada concepto y cada palabra define, habla, refiere a, ... un conjunto
 - *Ejemplos:*
 - *Este es tonto! o sea pertenece al conjunto de tontos, responde a las características del Tonto (compresión) o lo has puesto tú!!! (extension)*
 - *"- ¿A qué hora llegas?" o sea ¿Qué elemento del conjunto horas del día marcará el reloj cuando llegues?; "- A las -25 horas!", que no pertenece al conjunto de las horas del día; "- Éste es tonto!!!" o sea no conoce el conjunto de horas del día*



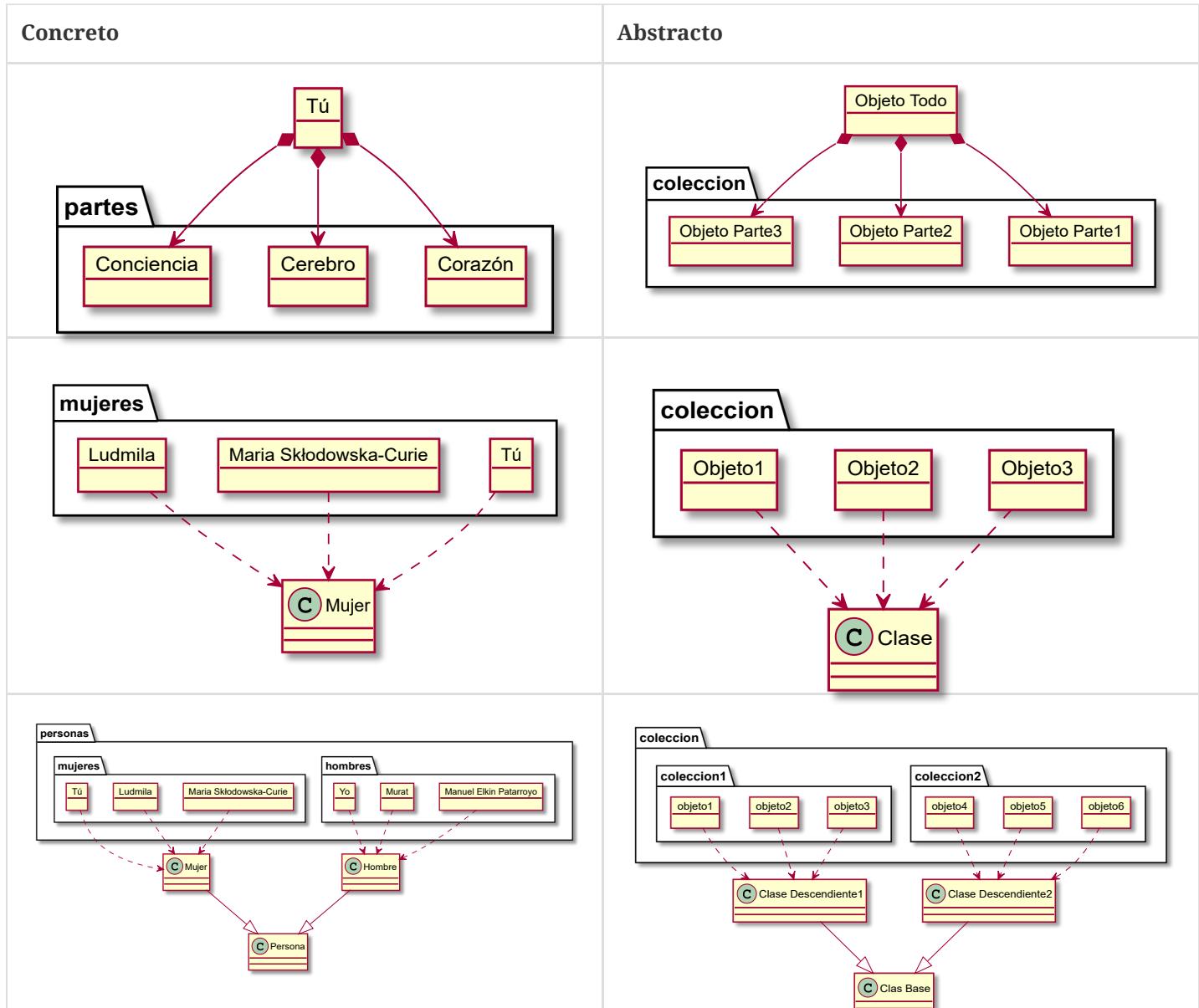
Tipos de Conjuntos

Extensión	Compresión
Colección desordenada de elementos no repetidos, asociados por extensión (aleatoriedad, capricho, ...)	Colección desordenada de elementos no repetidos, asociados por compresión (criterio, ecuación, condición, ...) con conjunto de características comunes, patrón
Ejemplos	Ejemplos
<i>las 7 maravillas del mundo antiguo: Chichén Itzá, en México; El Coliseo de Roma, en Italia; La estatua Cristo Redentor, en Brasil; La Gran Muralla China, en China; Machu Picchu, en Perú; Petra, en Jordania; El Taj Mahal, en India</i>	<i>mi cuerpo es el conjunto de mis órganos, huesos, ... dentro de mi piel</i>
<i>mis líderes favoritos: Sócrates, Lao Tse, Buda, Jesucristo, Mahoma, ...</i>	<i>la humanidad es el conjunto de todas las personas</i>
<i>alcaldes electos: resumen matemático, ley D `Hondt, del conjunto de caprichos, intuiciones, ... de cada votante</i>	<i>los números primos son el conjunto de números naturales cuyo conjunto de divisores tiene intersección vacía con el conjunto del 2 al anterior al número dado</i>

Tipos de Elementos

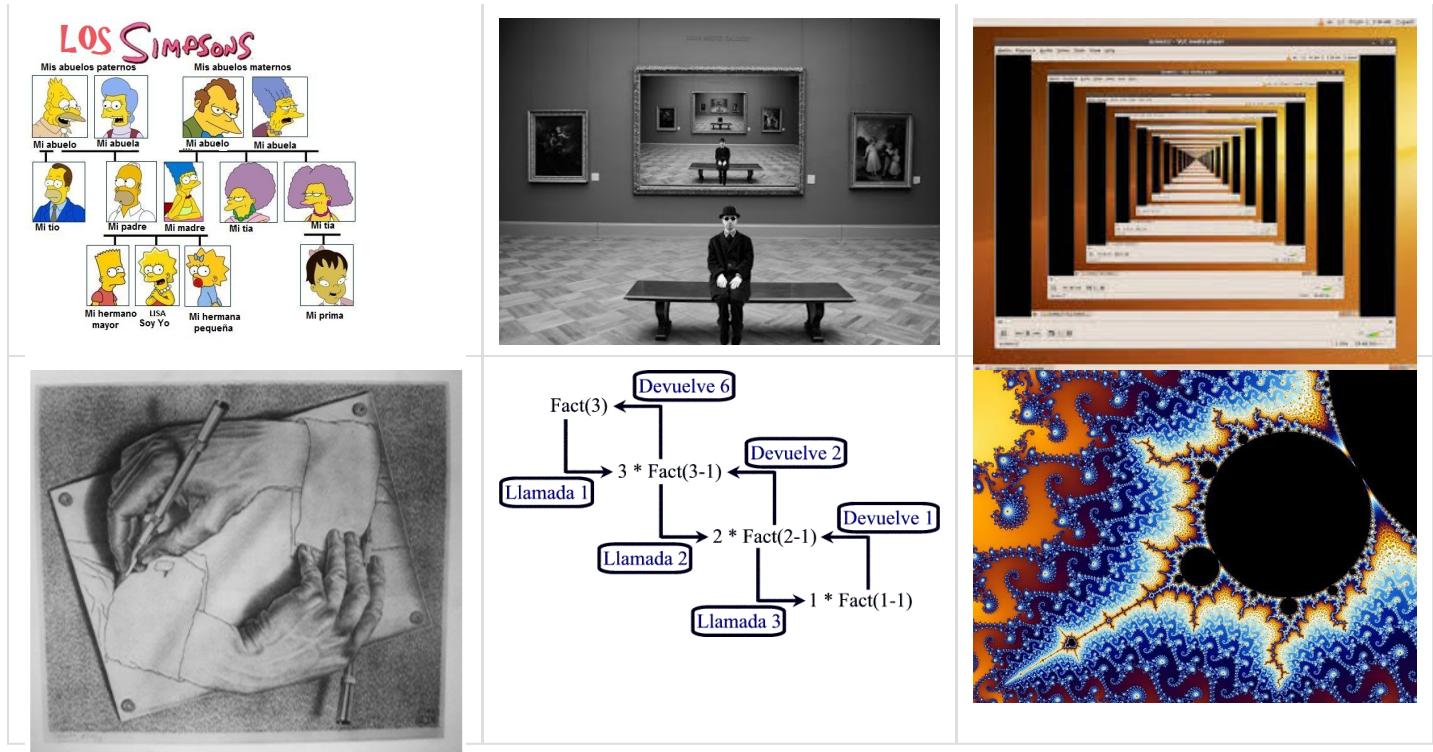
- Un **objeto** es un conjunto cuyos elementos son otros objetos
 - Sinónimos: **ente, cosa, archiperre, artilugio, adminículo, aparejo, pirindolo, ...**

- Una **clase** es un conjunto cuyos elementos son las características comunes de otro conjunto de objetos similares
 - Sinónimos: **tipo, entidad, clase, idea, ...**



Recursividad

- **Recursividad:** algo que se define/refiere, directa o indirectamente, sobre sí mismo
 - Vetada en el pasado por problemas con el infinito si no hay caso base:
 - Paradojas!!! Paradoja de Hércules y la Tortuga de Zenón,
 - Causa/Consecuencia ... la primera causa?!?! Dios!?! Más allá!?!
 - "Prohibida" por la Filosofía de la Ciencia hasta el siglo XIX: no puedes usar el concepto en la definición
 - Herramienta actual potentísima para la fundamentación de la lógica, matemática, programación, ...
 - Un número natural es el 0 o el sucesor de un número natural: 0 es 0, sucesor(0) es 1, sucesor(sucesor(0)) es 2, ...



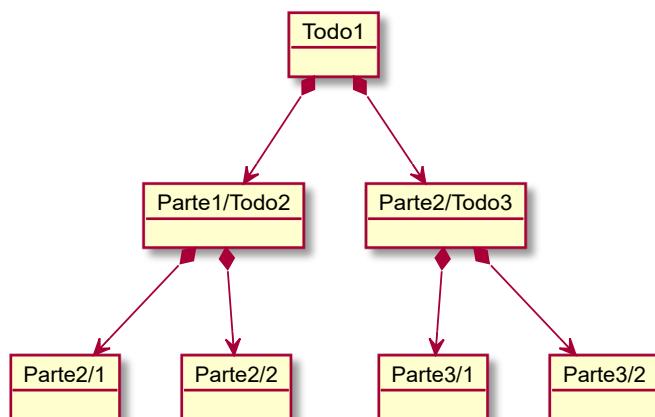
- Elementos recursivos

Condición	Concreto	Abstracto
Los elementos de un conjunto pueden ser conjuntos de otros elementos a su vez	<pre> graph TD Conjunto1[Conjunto1] --> Elemento1[Elemento1] Conjunto1 --> Conjunto3[Conjunto3] Conjunto1 --> Conjunto2[Conjunto2] Conjunto2 --> Elemento3[Elemento3] Conjunto2 --> Elemento2[Elemento2] Conjunto1 --> colección2["colección2"] Conjunto2 --> colección2 </pre>	<pre> classDiagram class Conjunto { <<Conjunto>> <<colección>> <<Elemento>> } class Elemento { <<colección>> } Conjunto "0..N" --> "1" Conjunto : colección Elemento "1" --> "0..N" Elemento : colección </pre>

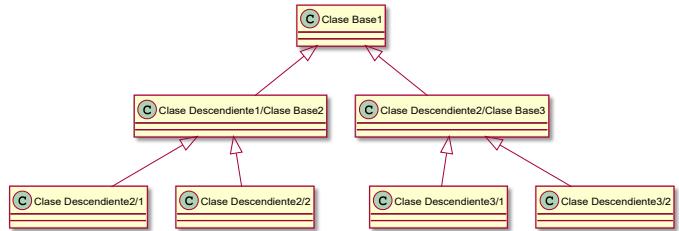
Jerarquías de Composición vs Clasificación

Jerarquía de Composición	Jerarquía de Clasificación
<ul style="list-style-type: none"> Surge de la unión de los objetos partes del todo que a su vez es parte de otro todo que a su vez ... ¿la parte es parte de el todo? 	<ul style="list-style-type: none"> Surge de la intersección de las características de las clases descendientes de la clase base que a su vez es descendiente de otra clase base que a su vez ... ¿la clase descendiente es una especialización de la clase base?

Jerarquía de Composición



Jerarquía de Clasificación



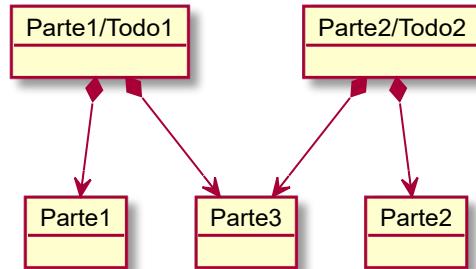
Ejemplo: tú eres el conjunto de tus órganos que cada uno es el conjunto de tejidos que son el conjunto de células, ... partículas elementales

Ejemplo: las personas son la unión del conjunto de las mujeres y los hombres con sus características generales, comunes, compartidas, ...

- Elementos compartidos

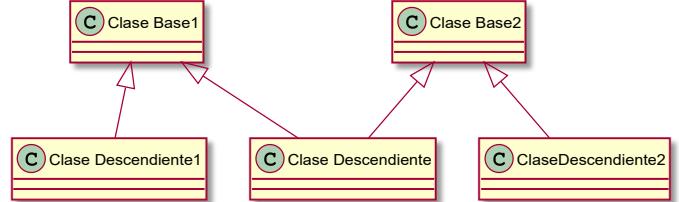
Jerarquía con Agregación

Un objeto parte pertenece a varios objetos todo



Jerarquía con Herencia Múltiple

Un clase descendiente comparte características con varias clases base

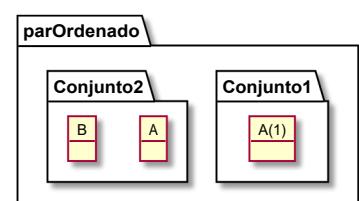


Yo pertenezco a la familia que me crió y a la que formé cuando crecí

Mi primo es español y francés, doble nacionalidad

Asociación

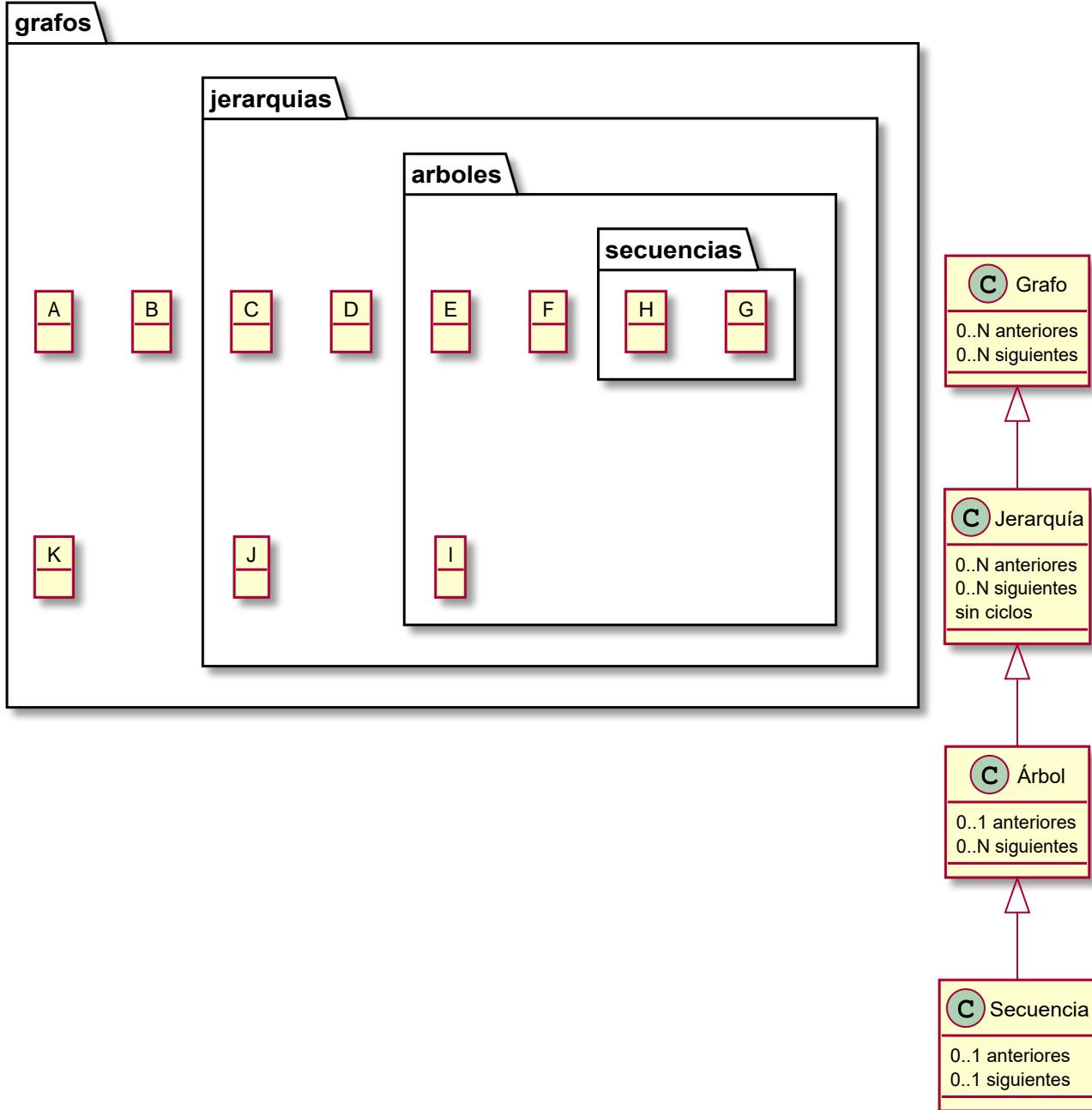
- Los conjuntos no incorporan orden pero se puede establecer, **característica emergente**
 - Par ordenado (a,b), a → b, ...
 - Un conjunto con dos conjuntos:
 - un subconjunto con los dos elementos
 - un subconjunto con el primer elemento
 - El primer elemento es la intersección de ambos subconjuntos
 - El segundo elemento es la diferencia de la unión y la intersección de ambos subconjuntos



Estructuras

Estructura	Condiciones	Ejemplo
Secuencia, lista, flujo, ...	<ul style="list-style-type: none"> 0..1 anterior, 0 para el primer elemento 0..1 siguiente, 0 para el último elemento 	
Árbol	<ul style="list-style-type: none"> 0..1 anterior, 0 para el elemento raíz 0..N siguientes, 0 para las hojas 	
Grafo, maraña, ...	<ul style="list-style-type: none"> 0..N anteriores, 0 para cualquier elemento 0..N siguientes, 0 para cualquier elemento 	
Jerarquía	<ul style="list-style-type: none"> 0..N anteriores, 0 para cualquier elemento 0..N siguientes, 0 para cualquier elemento sin ciclos 	

- Clasificación de Estructuras



Estrategias de Clasificación

Objetos	Clasificación	Jerarquía de Clasificación																														
<table border="1"> <tr> <td>o26 q=46 r=8 s=29</td> <td>o15 q=34 r=17 s=123</td> <td>o16 q=4 r=12 s=25</td> <td>o14 q=52 r=76 s=27</td> <td>o10 q=56 r=2 s=19</td> <td>o24 q=3 r=16 s=92</td> </tr> <tr> <td>o29 q=30 r=129 s=79</td> <td>o5 q=52 r=15 s=12</td> <td>o20 q=1 r=14 s=77</td> <td>o4 p=68 s=13 q=78</td> <td>o6 p=45 s=48 q=12</td> <td>o19 p=23 s=90 q=14</td> </tr> <tr> <td>o21 p=15 s=78 q=1</td> <td>o25 p=42 s=23 q=87</td> <td>o13 p=97 s=111 q=3</td> <td>o30 p=65 s=47 q=80</td> <td>o23 p=22 s=55 q=77</td> <td>o17 p=25 s=33 q=7</td> </tr> <tr> <td>o28 p=68 s=86 q=43p</td> <td>o9 p=37 s=10</td> <td>o2 p=70 o=11</td> <td>o3 p=19 o=5</td> <td>o8 p=51 o=81</td> <td>o12 p=31 o=6</td> </tr> <tr> <td>o27 p=93 o=103</td> <td>o1 p=26 o=72</td> <td>o18 p=48 o=39</td> <td>o22 p=75 o=55</td> <td>o11 p=62 o=12</td> <td>o7 p=74 o=40</td> </tr> </table>	o26 q=46 r=8 s=29	o15 q=34 r=17 s=123	o16 q=4 r=12 s=25	o14 q=52 r=76 s=27	o10 q=56 r=2 s=19	o24 q=3 r=16 s=92	o29 q=30 r=129 s=79	o5 q=52 r=15 s=12	o20 q=1 r=14 s=77	o4 p=68 s=13 q=78	o6 p=45 s=48 q=12	o19 p=23 s=90 q=14	o21 p=15 s=78 q=1	o25 p=42 s=23 q=87	o13 p=97 s=111 q=3	o30 p=65 s=47 q=80	o23 p=22 s=55 q=77	o17 p=25 s=33 q=7	o28 p=68 s=86 q=43p	o9 p=37 s=10	o2 p=70 o=11	o3 p=19 o=5	o8 p=51 o=81	o12 p=31 o=6	o27 p=93 o=103	o1 p=26 o=72	o18 p=48 o=39	o22 p=75 o=55	o11 p=62 o=12	o7 p=74 o=40		
o26 q=46 r=8 s=29	o15 q=34 r=17 s=123	o16 q=4 r=12 s=25	o14 q=52 r=76 s=27	o10 q=56 r=2 s=19	o24 q=3 r=16 s=92																											
o29 q=30 r=129 s=79	o5 q=52 r=15 s=12	o20 q=1 r=14 s=77	o4 p=68 s=13 q=78	o6 p=45 s=48 q=12	o19 p=23 s=90 q=14																											
o21 p=15 s=78 q=1	o25 p=42 s=23 q=87	o13 p=97 s=111 q=3	o30 p=65 s=47 q=80	o23 p=22 s=55 q=77	o17 p=25 s=33 q=7																											
o28 p=68 s=86 q=43p	o9 p=37 s=10	o2 p=70 o=11	o3 p=19 o=5	o8 p=51 o=81	o12 p=31 o=6																											
o27 p=93 o=103	o1 p=26 o=72	o18 p=48 o=39	o22 p=75 o=55	o11 p=62 o=12	o7 p=74 o=40																											

- Categorización clásica
- Agrupación conceptual
- Teoría de Prototipos

Categorización clásica

“ Todas las entidades que tienen una determinada propiedad o conjunto de propiedades en común forman una categoría(tipo/clase). Estas propiedades son necesarias y suficientes para definir la categoría ”

— Booch
96

- Las categorías naturales tienden a ser un poco incómodas: la mayoría de los pájaros vuelan, pero algunos no lo hacen; las sillas pueden consistir de madera, plástico o metal y pueden tener casi cualquier número de patas, dependiendo del capricho del diseñador.
- Parece prácticamente **imposible llegar a una lista de propiedades para cualquier categoría** natural que excluya a todos los ejemplos que no están en la categoría e incluya todos los ejemplos que se encuentran en la categoría
- Modelo de lenguajes de **programación orientados a objetos con clases**, como Java

Agrupación conceptual

“ Las clases se generan mediante la formulación de primeras descripciones conceptuales de estas clases y, a continuación, la clasificación de las entidades de acuerdo con las descripciones ”

— Stepp; Michalski

- Podemos afirmar un concepto como "canción de amor." Este es un concepto más que una propiedad, para la "canción de amor"-idad de cualquier canción no es algo que se pueda medir empíricamente. Sin embargo, si decidimos que una determinada canción es más una canción de amor que no, nos situamos en esta categoría. - Por lo tanto, la agrupación conceptual representa más una **agrupación probabilística de objetos**

- Problemas de objetividad en conceptos con excepciones (pingüino no vuela, negros de piel blanca, ...)

Teoría de Prototipos

“ Wittgenstein señaló que una categoría como juego no encaja en el molde clásico, ya que no hay propiedades comunes compartidos por todos los juegos. . . . Aunque no existe una única colección de propiedades que comparten todos los juegos, la categoría juego está unida por lo que Wittgenstein llama parecidos de familia. . . . Wittgenstein también observó que no había límite fijo en la categoría juego. La categoría podría extenderse y nuevos tipos de juegos ser introducidos, siempre que se parezcan a los juegos anteriores de manera apropiada

— Lakov

- Una clase de objetos está representada por un objeto prototípico y un objeto se considera que es un miembro de esta clase si, y sólo si se asemeja a este prototipo de forma significativa. Nosotros agrupamos cosas de acuerdo a distintos conceptos según el grado de su **relación con prototipos concretos**
 - Modelo de lenguajes de **programación orientados a objetos por prototipos**, sin clases, como JavaScript

Software

Santa Tecla
parqueNaturalSantaTecla@gmail.com
Version 0.0.1

Índice

Justificación: ¿Por qué?

Historia del Hardware

Definición: ¿Qué?

Objetivos: ¿Para qué?

Sistema de Información

Descripción: ¿Cómo?

Actores, Actividades y Artefactos

Paradigmas del Software

Sistema Complejo

Calidad del Software

Mantenibilidad

Viscosidad vs Fluidez

Rigidez vs Flexibilidad

Fragilidad vs Fortaleza

Inmovilidad vs Reusabilidad

Evolución del Software

Sencillez

Bibliografía

Justificación: ¿Por qué?

Historia del Hardware

- La humanidad gracias a sus herramientas y, en particular, al conocimiento (ciencias, ingenierías, ...), ha construido grandes sistemas artificiales: acueductos, telares con tarjetas perforadas, red eléctrica, red telefónica, ... para **reutilizar, automatizar y simplificar** tareas
 - 8000 aec., Los sumerios construyen telares para cubrirse
 - 2000 aec., Los sumerios utilizaban el **ábaco**, primera memoria
 - 1642 ec., **Blaise Pascal** construye la Pascalina, primera calculadora mecánica girando ruedas
 - 1801 ec., **Jacquard** construye el primer telar mecánico y automático con tarjetas perforadas para definir los dibujos
 - 1842 ec., **Charles Babbage** y **Ada Lovelace** trabajan sobre la **Máquina Analítica**, con las tarjetas perforadas de los telares ... pero no llegó a funcionar aunque Ada ya escribió las primeras líneas de código de la historia.
 - 1884 ec., **Hollerith** desarrolló la **Máquina Tabular** de tarjetas perforadas para ordenar el registro de propiedad en la Conquista del Oeste
 - 1936 ec., **Konrad Zuse**, ingeniero alemán, diseñó y fabricó la Z1, la que para muchos es la primera computadora programable de la historia

Objeto	Capacidad cualitativa	Capacidad cuantitativa
Ser humano	Muy buena : reconocimiento de patrones, asociaciones, recursividad, ...	Muy mala : errores por cansancio, desmotivación, ... y muy lentos
Hardware	Muy mala : ningún computador superó la prueba de Turing	Muy buena : sin errores y a toda velocidad

Definición: ¿Qué?

“ Software es la información que suministra el desarrollador a la computadora para que manipule de forma automática la información que suministra el usuario ”

— Brad Cox

- La información suministrada por el Desarrollador de Software es de **diversa naturaleza**:
 - Programas en lenguajes de programación (Java, C/C++, ...),
 - Scripts para la creación de las tablas de las bases de datos y su población (SQL),
 - Scripts para la generación de páginas dinámicas en aplicaciones Web (JSP, PHP,...),
 - Presentaciones en lenguajes de formato para aplicaciones Web (HTML, CSS, ...)
 - Datos de configuración en diversos formatos (texto libre, XML, JSON, ...)
 - Multimedia en formatos de imagen, sonido o video para elementos gráficos en la Interfaz de Usuario (*.png, *.waw, *.mpeg, ...)
 - ...

Objetivos: ¿Para qué?

- **Efectividad** en la **gestión de sistemas de información**, requeridos por los **usuarios**, con
 - **Eficacia**, sin errores en **cálculos**, filtrados, secuencias de acciones, ...
 - **Eficiencia**, con escaso consumo de recursos:
 - **hardware y energía eléctrica**
 - **tiempo de los usuarios** para aprender y **explotar** la gestión del sistema de información

Sistema de Información

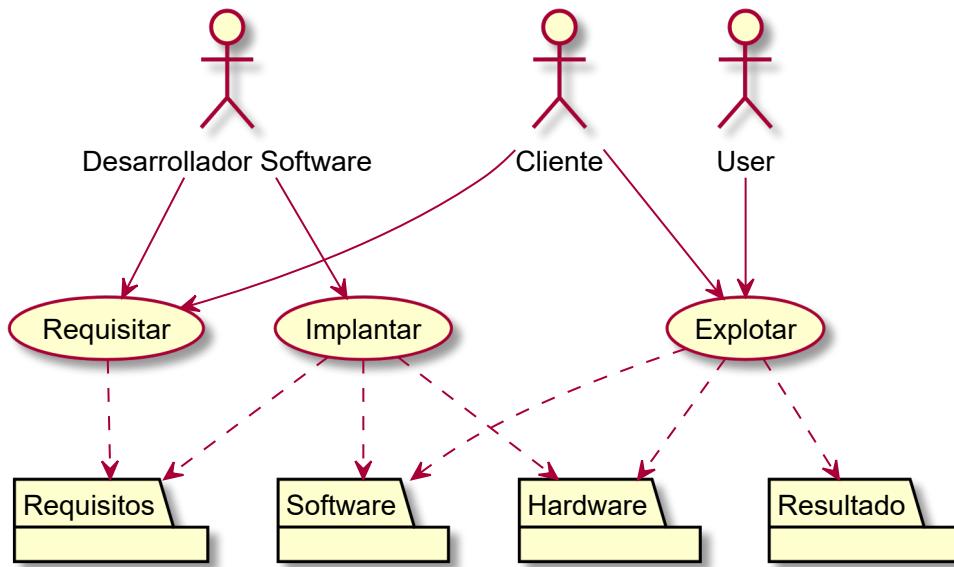
“Un sistema de información (SI) es un conjunto de elementos orientados al tratamiento y administración de datos e información, organizados y listos para su uso posterior, generados para cubrir una necesidad o un objetivo”

— Sistema Información
Wiki

- **Gestión (CRUD)**, es el tratamiento de la información, Informática!
 - **altas** (Create) de información en el sistema
 - **bajas** (Delete) de información en el sistema
 - **modificaciones** (Update) de información en el sistema
 - **consultas** (Read) de información en el sistema

Descripción: ¿Cómo?

Actores, Actividades y Artefactos



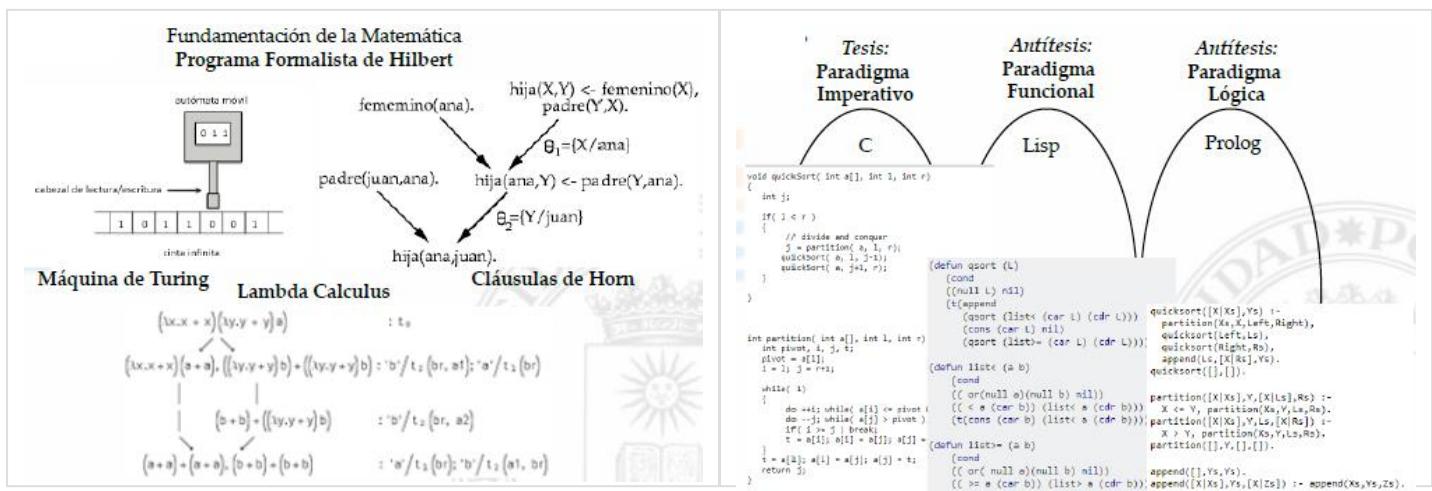
Paradigmas del Software

“...'ciencia normal' significa investigación basada firmemente en una o más realizaciones científicas pasadas, realizaciones que alguna comunidad científica particular reconoce, durante cierto tiempo, como fundamento para su práctica posterior. [...] Voy a llamar, de ahora en adelante, a las realizaciones que comparten esas dos características, 'paradigmas', término que se relaciona estrechamente con 'ciencia normal'. **Paradigma es un conjunto de prácticas y saberes que definen una disciplina científica durante un período específico**

— Thomas S. Kuhn
La estructura de las revoluciones científicas

- **Ejemplos:**

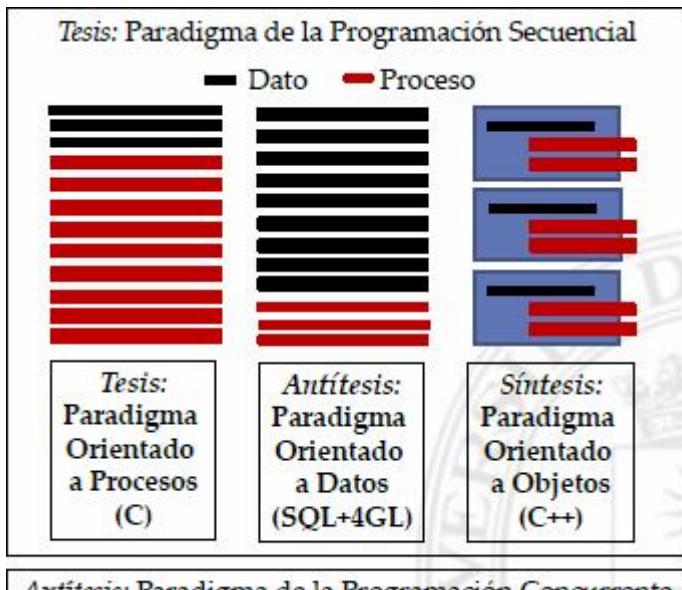
- la física de Newton tiene por axiomas que el espacio y el tiempo son dos cosas diferentes y absolutas mientras que la física de Einstein tiene por axioma un espacio-tiempo relativo que se curva ...
- un hippy y un yuppi tienen diferentes objetivos, valores, principios, métodos, ... en este mismo mundo
- Paradigmas de la Programación respecto a sus **fundamentos matemáticos**:



Paradigma	Fundamento	Enfoque	Lenguajes
Paradigma Imperativo	Máquina de Turing	asignación para cambios de estados de los datos variables	Cobol, Fortran, C, Ada, ...
Paradigma Funcional	Cálculo Lambda	Funciones matemáticas sobre valores constantes con el algoritmo de Correspondencia de Patrones para las reglas de sustitución	Lisp, FP, Scheme, ML, Haskell, ...
Paradigma Lógico	Cláusulas de Horn	Hechos y predicados lógicos con algoritmo de Unificación para las reglas de sustitución	Prolog, ...

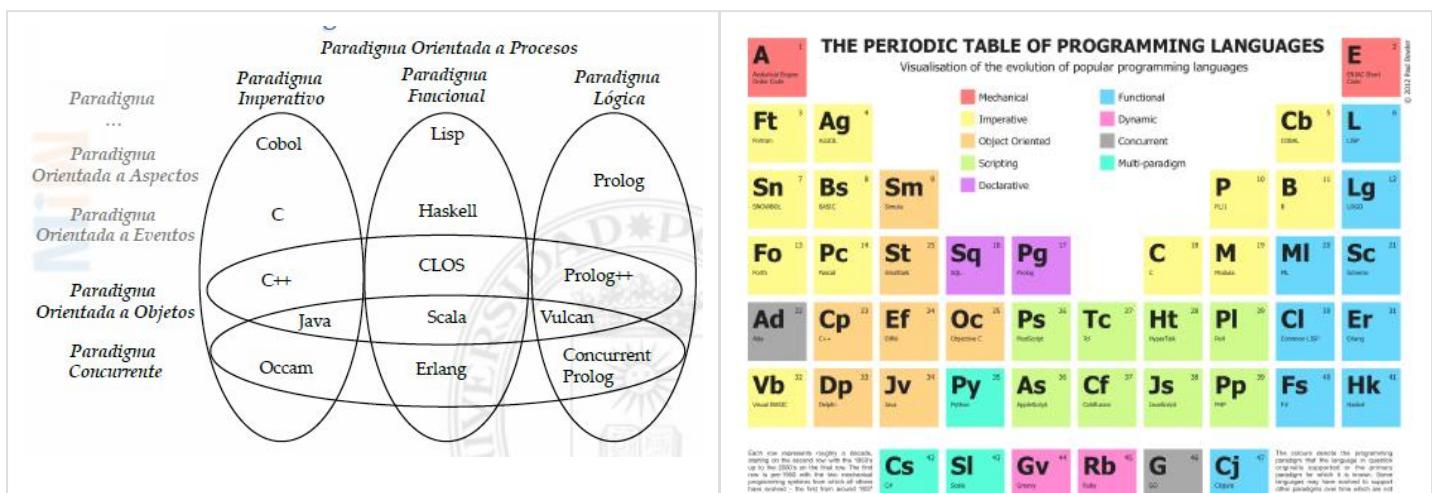
- Se llama conjuntamente **Paradigma Declarativo**, unión del Paradigma Funcional y Lógico, enfrentado al Paradigma Imperativo, porque carece de variables con estado y, por tanto, de bucles. Sus soluciones son eminentemente **recursivas**, declarando la solución de un problema como alguna operación sobre la solución del problema en un grado menos.
- Paradigmas de la Programación respecto a su **organización**:

Paradigma	Enfoque	Lenguajes
Paradigma Orientado a Procesos	Preponderancia de las operaciones sobre los datos	C, Pascal, ...
Paradigma Orientado a Datos	Preponderancia de los datos sobre las operaciones (1 ^a forma normal, 2 ^a forma normal, ..., Boyce-Codd)	SQL, ...
Paradigma Orientado a Objetos	Equilibrio entre datos y operaciones, conformando una clase de objetos reinstanciable	Smalltalk, C++, Java, CLOS, Python, ...



Antítesis: Paradigma de la Programación Concurrente

- Otros paradigmas de la Programación: Concurrente, Orientado a Eventos, a Aspectos, ... Lenguajes Exóticos: bidimensionales, ...!!!



Sistema Complejo

- **Sistema:** Software es un conjunto de clases/módulos relacionándose por herencia, composición, ... o interdependientes formando una aplicación. Cada aplicación está delimitada por su entorno tecnológico-comercial, descrito por su arquitectura del software y requisitos y expresado en su ejecución
 - **Sistema complejo:** Software de una aplicación media (~100.000 líneas de código) tiene una complejidad que excede la capacidad intelectual humana
 - **Características de Sistemas complejos:**
 - Estructura jerárquica gracias a sus jerarquías de herencia, composición, paquetes con clases con atributos y métodos, métodos con sentencias, sentencias con expresiones, ...
 - Elementos primitivos relativos gracias a sus tipos primitivos dependiendo del lenguaje (enteros, cadena de caracteres?, fechas?, ...) y los definidos por el usuario
 - Separación de asuntos gracias a la encapsulación y modularidad
 - Patrones comunes gracias a algunos métodos de clases que corresponden al paso de mensajes a objetos
 - Formas intermedias estables gracias a las metodologías iterativas

Calidad del Software

- Distintas características no funcionales del software
 - **Fiabilidad**, cumpla una determinada función bajo ciertas condiciones durante un tiempo determinado
 - **Usabilidad**, sencillo de usar porque facilita la lectura de los textos, descarga rápidamente la información y presenta funciones y menús sencillos, por lo que el usuario encuentra satisfechas sus consultas y cómodo su uso
 - **Accesibilidad**, pueda ser accedido y usado por el mayor número posible de personas, indiferentemente de las limitaciones propias del individuo o de las derivadas del contexto de uso
 - **Seguridad**, proteger los datos que tiene, maneja y dispone para preservar la confidencialidad, la integridad y la disponibilidad
 - **Confidencialidad**, acceso a la información mediante autorización y control para prevenir la divulgación no autorizada de la información
 - **Integridad**, para modificar la información mediante autorización
 - **Disponibilidad**, degradación en cuanto a accesos para prevenir interrupciones no autorizadas
 - **Interoperabilidad**, habilidad de dos o más sistemas o componentes para intercambiar información y utilizar la información intercambiada
 - **Portabilidad**, habilidad de reutilizar en vez de crear un nuevo software cuando se pasa de una plataforma a otra
 - **Escalabilidad**, habilidad para reaccionar y adaptarse sin perder calidad cuando aumentan el tamaño del sistema de información
 - **Extensibilidad**, habilidad de tener la posibilidad de extenderse con nuevas funcionalidades
 - ... todas dependen de:

Mantenibilidad

- **Mantenibilidad**, habilidad de conservar su funcionamiento normal o para restituirlo una vez se ha presentado un evento de falla o un nuevo requisito
 - **Mantenibilidad Correctiva**, para la eliminación de errores de cualquier otra cualidad
 - **Mantenibilidad Perfectiva**, para la modificación de su funcionalidad con cualquier otra cualidad
 - **Mantenibilidad Adaptativa**, para la modificación de su infraestructura para cualquier otra cualidad

Diagrama con relación a los actores

Mantensible	No mantensible
<i>Fluido</i>	Viscoso
<i>Flexible</i>	Rígido
<i>Fuerte</i>	Frágil
<i>Reusable</i>	Inmóvil

Viscosidad vs Fluidez

- Viscosidad viene en dos formas: viscosidad del diseño, y la viscosidad del entorno.
 - La **viscosidad del diseño** se produce cuando nos enfrentamos a un cambio, los ingenieros suelen encontrar más de una manera de hacer el cambio. Algunas de las formas conservan el diseño, otros no lo hacen, es decir, son atajos. Cuando **preservar el diseño es más difícil que emplear los atajos**, a continuación, la viscosidad del diseño es alta. Es fácil de hacer las cosas mal, pero difícil de hacer lo correcto.
 - La **viscosidad del entorno** se produce cuando el **entorno de desarrollo es lento e ineficiente**. Por ejemplo, si los tiempos de compilación son muy largos, los ingenieros tendrán la tentación de hacer cambios que no obligan a grandes re-compilaciones, a pesar de que esos cambios no son óptimos desde el punto de vista del diseño. Si el sistema de control de código fuente requiere horas para comprobar tan sólo unos pocos archivos, consecuentemente, los ingenieros tendrán la tentación de hacer cambios que requieren el menor número de subidas (commits) como sea posible, independientemente de si el diseño se conserva.

Rigidez vs Flexibilidad

- Rigidez es la tendencia del software que es **difícil de cambiar**, incluso en formas simples. Cada cambio provoca una **cascada de cambios posteriores** en los módulos dependientes. Lo que comienza como un simple cambio de dos días a un módulo se convierte en un maratón de varias semanas de cambios en el módulo después de otros módulos según los ingenieros persiguen el hilo del cambio a través de la aplicación.
 - Cuando el software se comporta de esta manera, los gerentes temen que permitirá a los ingenieros no solucionar problemas críticos. Esta resistencia se deriva del hecho de que ellos no saben, con confiabilidad, cuando terminarán. Si los gerentes insisten, los ingenieros se perderán en este tipo de problemas, que pueden **desaparecer durante largos períodos de tiempo**.
 - Cuando los miedos del gerente son tan agudos que se niegan a permitir cambios en el software, la rigidez oficial se instala. Por lo tanto, lo que comienza como una **deficiencia de diseño**, termina siendo una política de gestión adversa.

Fragilidad vs Fortaleza

- En estrecha relación con la rigidez está la fragilidad.
- Fragilidad es la tendencia del software para **estropearse en muchos lugares cada vez que se cambia**. A menudo, el error se produce en las zonas que no tienen ninguna relación conceptual con el área que se ha cambiado..
 - Según empeora la fragilidad, la probabilidad de error aumenta con el tiempo, asintóticamente acercándose 1. Este tipo de software es **imposible de mantener**. Cada solución hace que sea peor, la introducción de más problemas que soluciones.
 - Tales errores llenan las sensaciones de los gerentes de malos presagios. Cada vez que autorizan una solución, temen que el software va a estropearse de alguna manera inesperada. Este tipo de software hace que los gerentes y los clientes sospechen que los **desarrolladores han perdido el control de su software. La desconfianza reina, y la credibilidad se pierde**.

Inmovilidad vs Reusabilidad

- La inmovilidad es la **imposibilidad de volver a utilizar el software de otros proyectos o de partes del mismo proyecto**.
 - A menudo sucede que un ingeniero descubrirá que necesita un módulo que es similar a uno que escribió otro ingeniero. Sin embargo, también sucede a menudo que **el módulo en cuestión tiene demasiado equipaje del que depende**.
 - Después de mucho trabajo, los ingenieros descubren que el **trabajo y el riesgo requerido para separar las partes deseables del software de las partes no deseadas** son demasiado grandes como para tolerarlo. Y así, el software es simplemente reescrito en lugar de reutilizado.

Evolución del Software

“Ley del Decremento de la calidad: La calidad de los sistemas software comenzará a disminuir a menos que dichos sistemas se adapten a los cambios de su entorno de funcionamiento.

—Lehman y Belady

“Ley del Cambio Continuo: Un programa que se usa en un ámbito del mundo real, necesariamente debe cambiar o convertirse cada vez en menos útil y menos satisfactorio para el usuario.

—Lehman y Belady

“Ley de la Autorregulación: Los atributos de los sistemas, tales como tamaño, tiempo entre entregas y la cantidad de errores documentados son aproximadamente invariantes para cada entrega del sistema.

—Lehman y Belady

“Ley de la Conservación de la familiaridad: A medida que un sistema evoluciona todo lo que está asociado con ello, como los desarrolladores, personal de ventas, y usuarios por ejemplo, deben mantener un conocimiento total de su contenido y su comportamiento para lograr una evolución satisfactoria. Un crecimiento exagerado disminuye esta capacidad.

—Lehman y Belady

“Ley del Crecimiento continuado: La funcionalidad ofrecida por los sistemas tiene que crecer continuamente para mantener la satisfacción de los usuarios.

—Lehman y Belady

“Ley de la Complejidad Creciente: Debido a que los programas cambian por evolución, su estructura se convierte en más compleja a menos que se hagan esfuerzos activos para evitar este fenómeno

—Lehman y Belady

“Ley de la Estabilidad organizacional: Durante el tiempo de vida de un programa, su velocidad de desarrollo es aproximadamente constante e independiente de los recursos dedicados al desarrollo del sistema.

—Lehman y Belady

“Ley de la Retroalimentación del sistema: Los procesos de evolución incorporan sistemas de retroalimentación multiagente y multibucle y estos deben ser tratados como sistemas de retroalimentación para lograr una mejora significativa del producto.

—Lehman y Belady

- El software no se muere, se convierte en un zombi!!!

“La observación general es que el principal enemigo de la fiabilidad, y tal vez de la calidad del software en general, es la complejidad

—Meyer

Cuanto más complejo sea un sistema, más abierto está al colapso total. Gran parte de la complejidad que se tiene que dominar es la complejidad arbitraria

—Booch

Sencillez

“El descubrimiento de un *orden* no es tarea fácil. . . sin embargo, una vez que el orden ha sido descubierto no hay dificultad alguna en reconocerlo”

— Descartes

“En igualdad de condiciones, la explicación más sencilla suele ser la correcta”

— Navaja de Occam

Antónimos	Antonyms	Libro	Autor
Mantenlo sencillo, estúpido!	Keep it simple, stupid!	En un portaviones?!?	Kelly Jhonson
Mantenlo pequeño y sencillo!	Keep it short and simple		
Mantenlo pequeño y simple!	Keep it small and simple		
Comprender el algoritmo	Understand the Algorithm	Smell Code (Clean Code)	Robert Martin

Antónimos	Antonyms	Libro	Autor
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

“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

- 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
- **Lo Sencillo es Equilibrado (Proporcional), Geométrico (Simétrico), ... con una estructura que reusa unos pocos patrones iterativos, recurrentes y recursivos!**

- 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 algoritmos muy eficientes** cuando no hay necesidad

Ingeniería del Software

Santa Tecla
parqueNaturalSantaTecla@gmail.com
Version 0.0.1

Índice

Justificación: ¿Por qué?

Crisis del Software

Economía del Software

Tiempo

Ámbito

Coste

Calidad

Interrelación

Calidad del Software

Definición: ¿Qué?

Objetivos: ¿Para qué?

Descripción: ¿Cómo?

Sistema Complejo

La complejidad del dominio del problema

Las limitaciones de la capacidad humana

La posible flexibilidad a través del software

El comportamiento de los sistemas discretos

La dificultad de gestionar el proceso de desarrollo

Disciplinas

Requisitos

Análisis

Diseño

Implementación

Pruebas

Despliegue

Proceso de Desarrollo Software

Proceso de Desarrollo Software en Cascada

Proceso de Desarrollo Software Iterativo

Bibliografía

Justificación: ¿Por qué?

Crisis del Software

- La Crisis del Software es la incapacidad para dominar la complejidad de los proyectos software que
 - Provocó la reunión en la *OTAN* en 1968 por parte de *Bauer, Dijkstra, Backus, Naur, ..., Wirth, Zimmermann!!!* que comenzaron la definición del lenguaje ALGOL, inspirador de otros muchos! **se entregan tarde, con deficiencias funcionales y por encima del presupuesto**, conocidos como **proyectos fracasados o problemáticos**.
 - Incluyendo **accidentes** que conllevaron a la muerte de tres personas en la máquina de radioterapia Therac-25 que emitió una sobredosis masiva de radiación u otros con pérdidas multimillonarias

Motivos de Proyectos Fracasados/Problemáticos	Incidencia
<ul style="list-style-type: none"> • Gestión <ul style="list-style-type: none"> ◦ Falta del involucración del usuario ◦ Poco apoyo de las gerencias involucradas ◦ Falta de recursos ◦ Tiempos poco realistas 	<ul style="list-style-type: none"> • 31.0% ◦ 12.8% ◦ 7.5% ◦ 6.4% ◦ 4.3%
<ul style="list-style-type: none"> • Requisitos <ul style="list-style-type: none"> ◦ Requerimientos y especificaciones poco claras ◦ Cambio de requerimientos y especificaciones ◦ Expectativas poco realistas ◦ Objetivos poco claros 	<ul style="list-style-type: none"> • 35.3% ◦ 12.3% ◦ 11.8% ◦ 5.9% ◦ 5.3%
<ul style="list-style-type: none"> • Tecnologías <ul style="list-style-type: none"> ◦ Tecnología deficiente ◦ Nuevas tecnologías 	<ul style="list-style-type: none"> • 10.7% ◦ 7.0% ◦ 3.7%
<ul style="list-style-type: none"> • Otros 	<ul style="list-style-type: none"> • 23.0%

- Estadísticas de *Standish Group* sobre 50.000 proyectos



“ *La complejidad del software es una propiedad esencial, no un accidente. Por esencial queremos decir que podemos dominar esta complejidad, pero nunca podemos hacer que se vaya. [...] A menudo llamamos esta condición la crisis del software, pero, francamente, una enfermedad que se ha llevado a los largo de este tiempo debe ser llamado normalidad*

— Brooks

Economía del Software

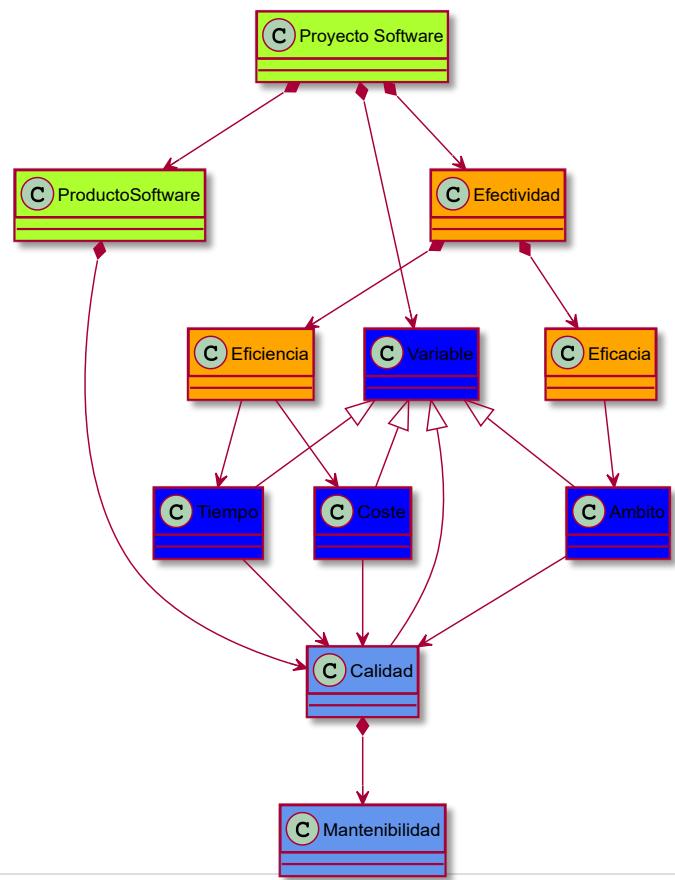
- Cuatro variables interrelacionadas
 - Tiempo
 - Ámbito
 - Coste
 - Calidad

“Nueve mujeres no pueden tener un bebé en un mes

— Brooks

Dieciocho mujeres aún no pueden tener un bebé en un mes

Variables de un Proyecto Software



Tiempo

- Las restricciones de controlar proyectos controlando el tiempo, generalmente vienen de fuera, de las manos del cliente
 - Si damos a un proyecto poco tiempo, la **calidad sufre, con el ámbito**.
 - **Disponer de más tiempo** para la entrega puede mejorar la calidad e incrementar el ámbito.
 - Ya que la realimentación desde los sistemas en producción es de mayor calidad que cualquier otra clase de realimentación, dar a un proyecto **demasiado tiempo será perjudicial**.

“Si la mayoría de los proyectos de tu organización son obsesivamente cortos, proyectos conducidos por el calendario, hay algo muy, muy malo. Cambios radicales en la organización del proceso de desarrollo software son necesarios, antes de que la compañía o su gente se arruine.

— Booch
Object Solutions

Ámbito

- Es **la más importante** a tener en cuenta
- **Naturaleza del Ámbito**
 - Poco ámbito permite entregar más rápido, mas calidad (mientras el problema del cliente esté resuelto) y más barato
 - Ámbito muy variable (**Ley del Cambio Continuo**):

- Porque los programadores y el **personal del negocio no tienen más que una idea vaga** sobre lo que tiene valor en el software que se está desarrollando.
- Porque los **requisitos nunca están claros al principio** y los clientes no pueden decirnos exactamente lo que quieren. El desarrollo de una pieza de software cambia sus propios requisitos ya que tan pronto como el cliente ve la primera versión, aprenden lo que quieren para la segunda versión ... o lo que realmente querían en la primera. Y esto es un aprendizaje valioso, porque no hay posibilidades de especulación. Este aprendizaje solamente puede venir de la experiencia. Pero los clientes no pueden estar solos, necesitan gente que pueda programar, no como guías, sino como compañeros.

• Gestión del Ámbito

- Si se gestiona activamente el ámbito, se puede proporcionar a los directores de proyecto y clientes **control sobre el coste, calidad y tiempo**.
- Intentando **no hacer demasiado**, mantenemos nuestra capacidad de producir la calidad requerida en un tiempo determinado.
- **Eliminación del ámbito** es una de las decisiones más importantes en la gestión del proyecto
 - Si el **tiempo está limitado por la fecha de lanzamiento** de una versión, hay siempre algo que podemos diferir a la siguiente versión.
 - Si dejas **fuerza importante funcionalidad al final de cada ciclo de versión**, el cliente quedará disgustado. Para evitar esto, se utilizan dos estrategias:
 - Implementa en **primer lugar los requisitos más importantes** del cliente, de tal manera que si se deja después alguna funcionalidad, es menos importante que la funcionalidad que ya está incorporada al sistema
 - Consigue **mucho práctica haciendo estimaciones** y realimentando los resultados reales. Mejores estimaciones reducen la probabilidad de que tengas que dejar fuera funcionalidad

Coste

- Al comienzo de un proyecto no puedes gastar mucho, la inversión tiene que comenzar siendo **pequeña y crecer con el tiempo**. Después, se puede de forma productiva gastar más y más dinero.
- **Mucho dinero** puede engrasar la maquinaria un poco, pero demasiado dinero pronto crea **más problemas que resuelve**. Mayores costes a menudo alimentan **objetivos tangenciales**, como estatus o prestigio (- "Tengo un proyecto de 150 personas!" - y respira profundamente")
- Dentro del **rango de inversión que pueda sensatamente hacerse**, gastando más dinero puedes aumentar el ámbito, o puedes intentar de forma más deliberada aumentar la calidad, o puedes (hasta cierto punto) reducir el tiempo de salida al mercado. También puede reducir las desavenencias: máquinas más rápidas, más especialistas técnicos, mejores oficinas.
- **Muy poco dinero**, no permite resolver el problema del negocio del cliente
 - Todas las restricciones sobre el coste pueden volver locos a los directores de proyecto. Especialmente si están sujetos a un proceso de presupuesto anual, están tan acostumbrados a considerarlo todo desde la perspectiva del coste y cometerán **grandes errores** al ignorar las restricciones sobre cuánto control te proporciona el coste.

Calidad

- Hay una extraña relación entre la calidad interna (que miden los programadores) y externa (que mide el cliente).
- **Sacrificar temporalmente la calidad interna** para reducir el tiempo de salida al mercado del producto, con la esperanza que la calidad externa no se vea muy dañada es tentador a corto plazo. Y puedes con frecuencia hacerlo impunemente generando una **confusión en cuestión de semanas o meses**. Al fin y al cabo, los

problemas de calidad interna te alcanzan a ti y hacen que tu software sea prohibitivamente caro de mantener.

- A menudo, al insistir en la **mejora de la calidad** puedes hacer que el proyecto esté listo en **menos tiempo**, o puedes conseguir **hacer más en un una cantidad de tiempo dada**. Se trabaja mejor si no se desmoraliza al producir software basura.

Interrelación

- No hay una relación sencilla entre las cuatro variables.
 - *Por ejemplo, no puedes obtener software más rápido, gastando más dinero*

“La forma de hacer en este modelo del juego del desarrollo del software es que las fuerzas externas (clientes, directores de proyecto) eligen los valores de tres variables cualquiera. El equipo de desarrollo determina el valor resultante de la cuarta variable”

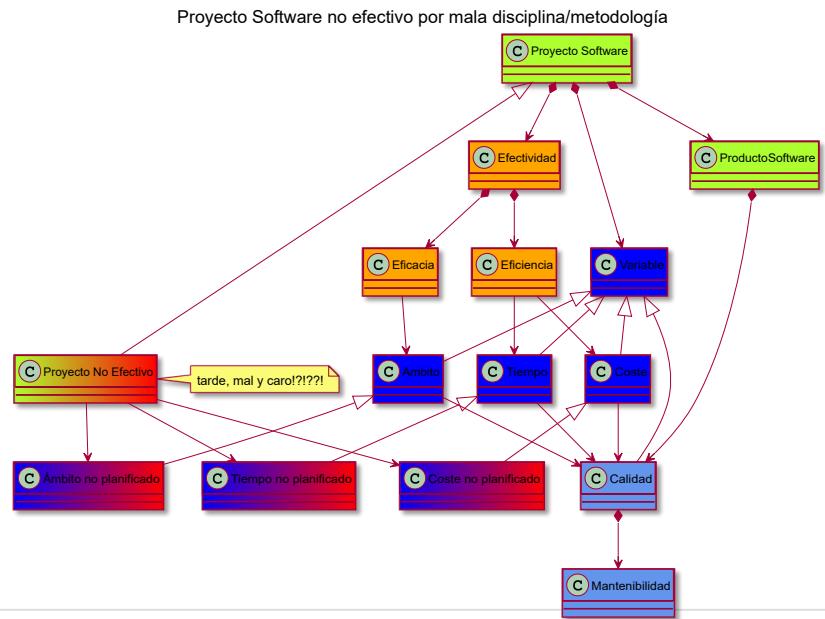
— Beck
1999

“Algunos directores de proyecto y clientes creen que pueden escoger el valor de las cuatro variables. Cuando esto suceda, la calidad siempre desaparecerá, ya que nadie hace bien el trabajo cuando está sujeto a una fuerte presión. También, probablemente, el tiempo estará fuera de control”

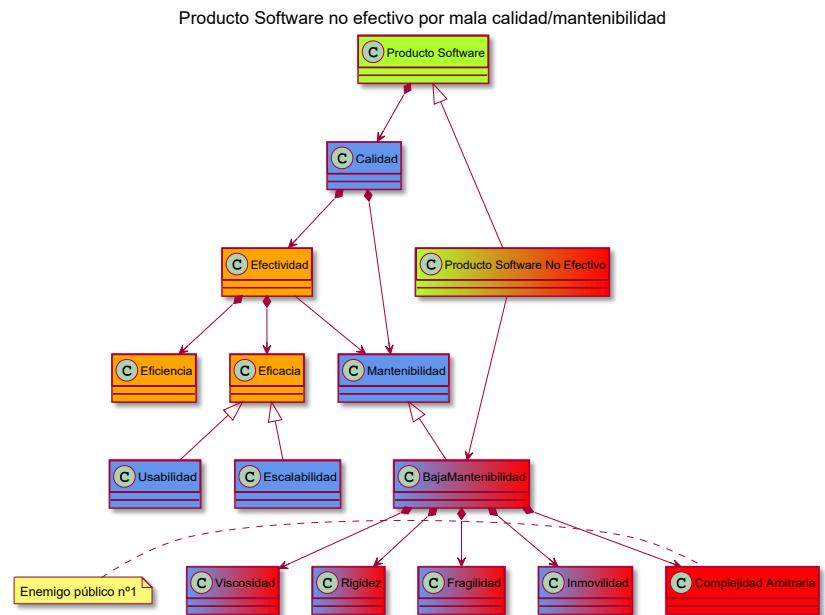
— Beck
1999

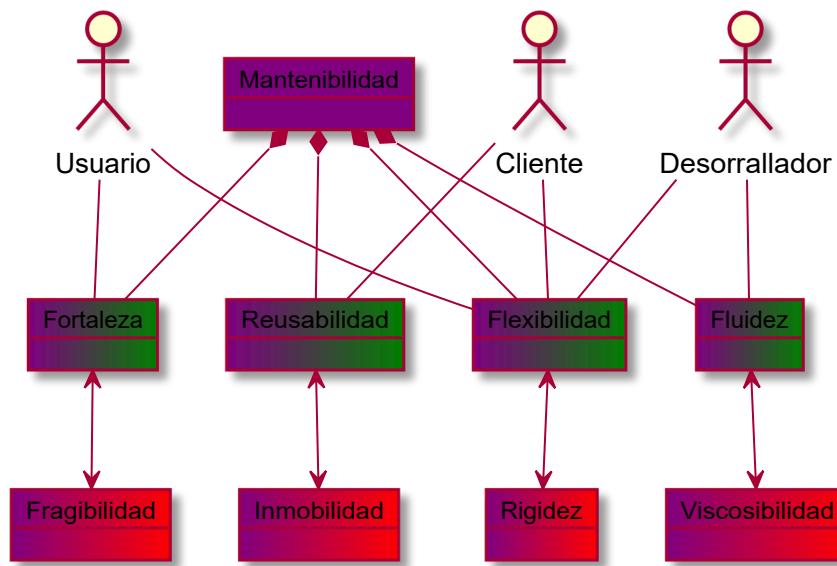
Calidad del Software

- 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
 - **inmóvil**, porque no se puede reutilizar con facilidad





Definición: ¿Qué?

“ Ingeniería de software es la aplicación práctica del conocimiento científico al diseño y construcción de programas de computadora y a la documentación asociada requerida para desarrollar, operar y mantenerlos. Se conoce también como **desarrollo de software** o **producción de software**

— Bohem
1976

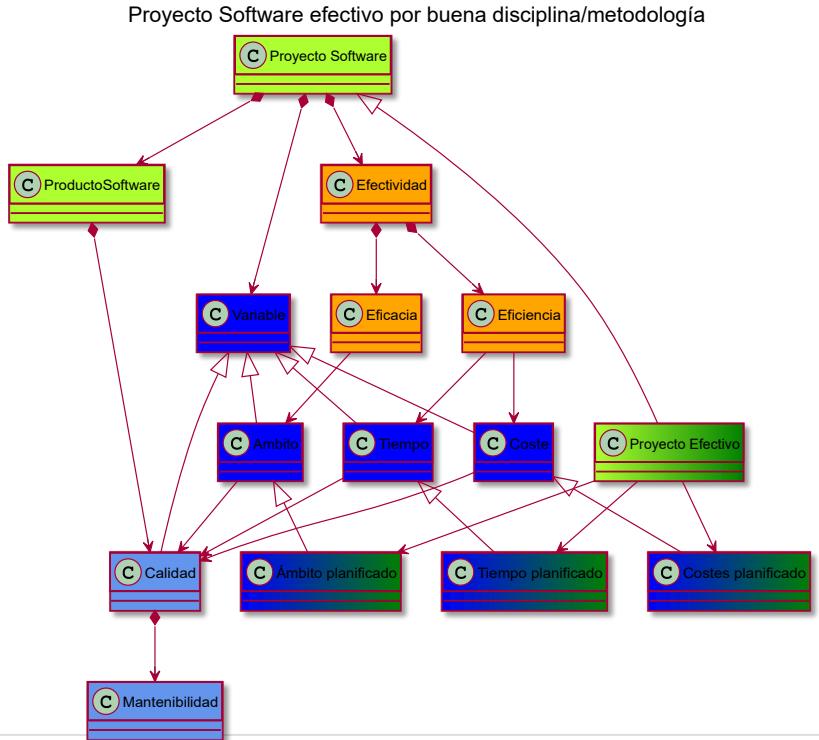
“ Proceso de Desarrollo Software: el conjunto total de **actividades necesarias para transformar los requerimientos del cliente en un conjunto consistente de artefactos que representan un producto software** y, en un momento posterior, para **transformar los cambios de estos requerimientos en una nueva versión** del producto software.

— Booch
1985

Ingeniería	Actividad	Ordenación
<p>“ El software es sagrado — Booch ... y requiere de un ritual</p>	<ul style="list-style-type: none"> • Requisitos • Análisis • Diseño • Implementación • Pruebas • Despliegue • Ecosistema • Gestión 	<ul style="list-style-type: none"> • Cascada • Iterativas <ul style="list-style-type: none"> ◦ Pesadas ◦ Ligeras • Artesanía del Software (Craftsmanship)

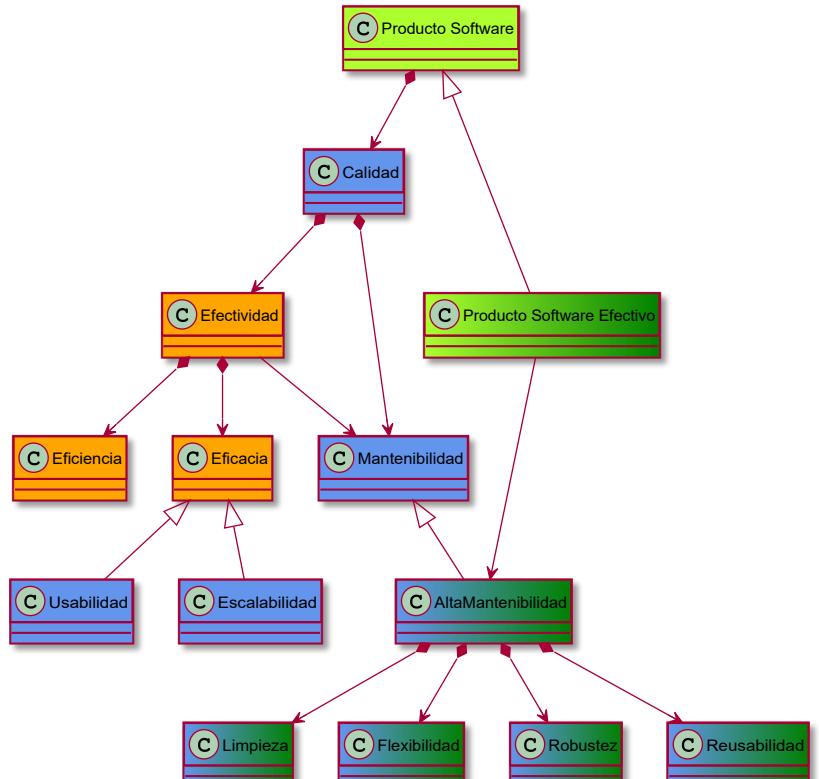
Objetivos: ¿Para qué?

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



- Producto Software efectivo
 - porque tiene **buenas variables**
 - **Es eficiente**
 - **Es eficaz en corrección, usabilidad, escalabilidad, ...**
 - porque tiene **buenas variables**
 - **fluido**, porque sí se puede entender con facilidad
 - **flexible**, porque sí se puede cambiar con facilidad
 - **fuerte**, porque sí se puede probar con facilidad
 - **reusable**, porque sí se puede reutilizar con facilidad

Producto Software efectivo por buena calidad/mantenibilidad



Descripción: ¿Cómo?

Sistema Complejo

El desarrollo de un proyecto software tiene una complejidad particular por las siguientes razones:

La complejidad del dominio del problema

- Los problemas que tratamos de resolver en el software a menudo implican elementos de complejidad ineludible, en las que encontramos una gran variedad **requisitos que pueden ser contradictorios, ambiguos u omitidos**.
- Una complicación adicional es que los requisitos de un sistema de software a menudo cambian durante su desarrollo (**Ley del Cambio Continuo** que conduce a la **Ley de la Complejidad Creciente**)

Las limitaciones de la capacidad humana

- **Número mágico de Miller** (7+2), Ley de Shyk

La posible flexibilidad a través del software

- **No existen muchos estándares en la industria del software** como en la industria de la construcción que tiene códigos y estándares para la calidad de las materias primas de construcción.
 - Como resultado, el desarrollo de software sigue siendo una empresa de trabajo intensivo.

El comportamiento de los sistemas discretos

- Dentro de una aplicación grande, puede haber cientos o incluso miles de variables, así como más de un hilo de control. Toda **la colección de estas variables, sus valores actuales y su dirección actual** y la pila de llamadas de cada proceso dentro del sistema constituyen el **estado actual de la aplicación**.
 - Por desgracia, es absolutamente imposible para una sola persona realizar un seguimiento de todos estos detalles a la vez. Este es el problema de la **caracterización del comportamiento de los sistemas discretos**.

La dificultad de gestionar el proceso de desarrollo

- La **gran cantidad de requisitos** de un sistema a veces es inevitable y nos obliga a escribir una **gran cantidad de software** nuevo o volver a utilizar el software existente en formas novedosas.
 - *Hace tan sólo unas décadas, los programas en lenguaje ensamblador de sólo unos pocos miles de líneas de código subrayaron los límites de nuestras capacidades de ingeniería de software.*
 - *Hoy en día, no es raro encontrar sistemas entregados cuyo tamaño se mide en cientos de miles o incluso millones de líneas de código, y todo eso en un lenguaje de programación de alto nivel*
 - *Analogía: El Quijote de la Mancha, unas 300.000 palabras, escrito por 8 personas a la vez en 6 meses a partir de la idea de otra persona! frente a los 18 años que tardó en escribirlo Miguel de Cervantes*

Problemas por la Complejidad del Desarrollo Software	Disciplina de la Ingeniería del Software
La complejidad del dominio del problema	Requisitos
Las limitaciones de la capacidad humana para el tratamiento de la complejidad	Mantenibilidad del Software del Análisis y Diseño
La posible flexibilidad a través del software	Reusabilidad del Software del Análisis y Diseño

Problemas por la Complejidad del Desarrollo Software	Disciplina de la Ingeniería del Software
Los problemas de la caracterización del comportamiento de los sistemas discretos	Pruebas y Despliegue
La dificultad de gestionar el proceso de desarrollo	Gestión

Disciplinas

Requisitos

- La disciplina de requisitos es el flujo de trabajo, incluyendo actividades, trabajadores y documentos, cuyo propósito principal es **dirigir el desarrollo hacia el sistema correcto al describir los requisitos del sistema así que pueda alcanzarse un acuerdo entre los clientes, usuarios y desarrolladores sobre lo que el sistema debería hacer:**
 - Establecer y mantener el acuerdo entre los clientes y otros interesados (stakeholders – gerencia, marketing, usuarios, ...) sobre lo que el sistema debería hacer
 - Proveer a los desarrolladores del sistema con una mejor comprensión de los requisitos del sistema
 - Definir los límites del sistema
 - Proveer las bases para planificar los aspectos técnicos del desarrollo
 - Proveer las bases para estimar los costes y tiempos para desarrollar el sistema

Análisis

- La disciplina de análisis es el flujo de trabajo, incluyendo trabajadores, actividades y documentos, cuyo principal objetivo es **analizar los requisitos a través de su refinamiento y estructura para realizar una compresión más precisa de los requisitos, una descripción de los requisitos que es fácil de mantener y ayuda a estructurar el sistema:**
 - Dar una especificación más precisa de los requisitos obtenidos en la captura de requisitos
 - Describir usando el lenguaje de los desarrolladores y poder introducir más formalismo y ser utilizado para razonar sobre el funcionamiento interno del sistema
 - Estructurar los requisitos de manera que facilite su comprensión, cambiándolos y, en general, mantenerlos
 - Acercarse al diseño, aunque sea un modelo en sí mismo, y es por tanto un elemento esencial cuando el sistema está conformado en diseño e implementación

Requisitos	Análisis
Describo usando el lenguaje del cliente	Describo usando el lenguaje de los desarrolladores (diagramas de clases)
Visión externa del sistema	Visión interna del sistema
Estructurado por requisitos , da estructura a la vista externa	Estructurado por clases estereotipadas y paquetes , da estructura a la vista interna
Usado principalmente como contrato entre los clientes y los desarrolladores sobre lo que el sistema debería hacer	Usado principalmente por desarrolladores para comprender qué forma debería tener el sistema
Contiene muchas redundancia, inconsistencias, .. entre los requisitos	No debería contener redundancias, inconsistencias, ... entre los requisitos
Captura la funcionalidad del sistema , incluyendo funcionalidad arquitectónica significativa	Esboza cómo realizar la funcionalidad en el sistema , incluyendo la funcionalidad arquitectónica significativa;

Diseño

- La disciplina de diseño es el flujo de trabajo, incluyendo actividades, trabajadores y documentos, cuyo principal propósito es **desarrollar enfocados en los requisitos no funcionales y en el dominio de la solución para preparar para la implementación y pruebas del sistema:**
 - Adquirir una comprensión profunda sobre los aspectos de los requisitos no funcionales y limitaciones relacionadas con:
 - los lenguajes de programación,
 - la reutilización de componentes,
 - sistemas operativos,
 - tecnologías de distribución y concurrencia,
 - tecnologías de bases de datos,
 - tecnologías de interfaz de usuario,
 - tecnologías de gestión de transacciones,
 - y así sucesivamente

Análisis	Diseño
Modelo conceptual porque es una abstracción del sistema y evita cuestiones de implementación	Modelo físico porque es un esbozo de la implementación
Menos formal	Más formal
Diseño genérico , aplicable a varios diseños concretos	No es genérico sino específico para una implementación
Tres estereotipos conceptuales en las clases: modelo, vista, controlador	Cualquier número de estereotipos físicos en las clases, dependiendo del lenguaje de implementación
Menos costoso para el desarrollo (1:5 frente al diseño)	Más costoso para el desarrollo (5:1 frente al análisis)
Pocas capas arquitectónicas	Muchas capas arquitectónicas
Puede no ser mantenido a través de todo el ciclo de vida del software	Debería ser mantenido a través de todo el ciclo de vida del software
Principalmente creado en trabajo de campo, talleres y similares	Principalmente creado por “ programación visual ” (ingeniería directa e inversa)
Define la estructura que es la entrada esencial para dar forma al sistema , incluyendo la creación del modelo de diseño	Dar forma al sistema mientras intenta preservar la estructura definida por el modelo de análisis
Enfatiza la investigación del problema y sus requisitos	Enfatiza en la solución conceptual que cubra los requisitos más que en su implementación
Haz lo correcto	Hazlo correctamente

Implementación

- La disciplina de implementación es el flujo de trabajo, incluyendo actividades, trabajadores y documentación, cuyo principal propósito es **implementar el sistema en términos de componentes, p.ej. código, scripts, ficheros binarios, código ejecutables:**
 - Definir la organización del código en términos de subsistemas de implementación organizados en capas
 - Implementar las clases y objetos en términos de componentes
 - Probar el desarrollo de componentes como unidades
 - Integrar en un sistema ejecutable el resultado producido por implementadores individuales o equipos

Pruebas

- La disciplina de pruebas es el flujo de trabajo, incluyendo actividades, trabajadores y documentación, cuyo principal propósito es **comprobar el resultado de la implementación al probar cada versión, incluyendo internas e intermedias, y versiones finales del sistema a entregar:**
 - Encontrar y documentar fallos en el producto software: defectos, problemas, ...
 - Avisar a la gestión sobre la calidad del software percibida
 - Evaluar las asunciones hechas en el diseño y especificación de requisitos a través de demostraciones concretas
 - Validar que el software trabaja como fue diseñado
 - Validar que los requisitos son implementados apropiadamente

Despliegue

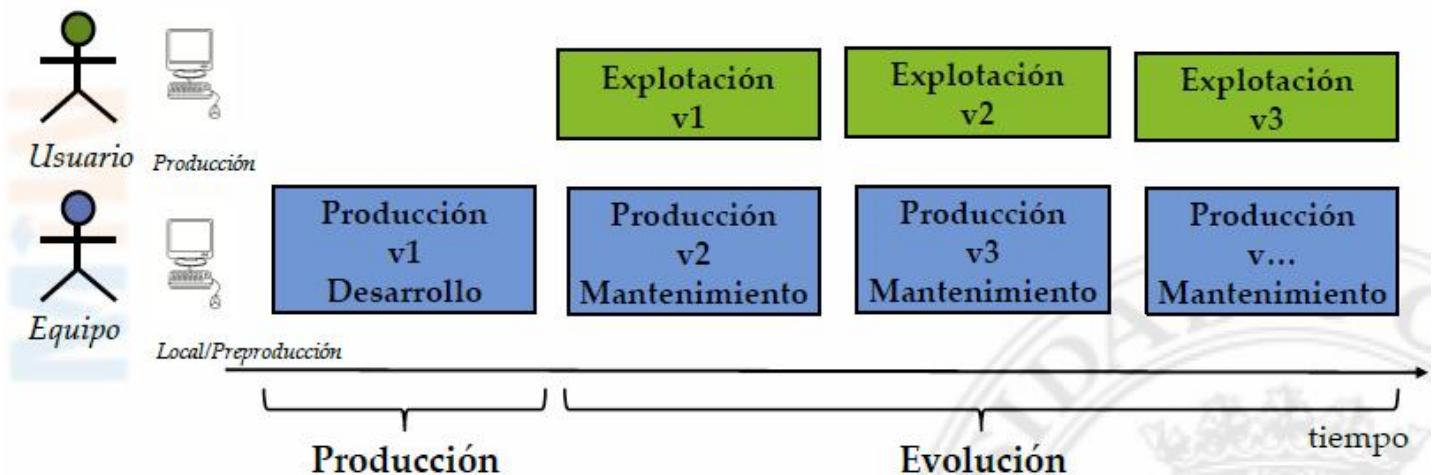
- La complejidad del software justifica la necesidad de **herramientas que aceleren su producción, controlen su calidad y monitoricen su gestión a lo largo de todas las disciplinas de la ingeniería del software**
- El ecosistema es un conjunto de servicios integrados orientados al desarrollo de software y su objetivo es mejorar la coordinación y el trabajo realizado por el equipo de desarrollo.

Disciplina	Necesidad	Herramientas
Requisitos	Se requiere un entorno colaborativo con editores, historial, autoría, respaldos, ... donde los especificadores de requisitos (casos de uso / historias de usuario) puedan escribir y el resto del equipo de desarrollo (analistas/diseñadores, programadores, probadores y desplegadores) puedan leer dichos requisitos centralizados	<i>Wiki de GitHub</i>
Análisis y Diseño	Se requiere de una herramienta CASE *(Computer Aided Software Engineering) que facilite la edición de *diagramas de análisis y diseño (diagramas de casos de uso, clases, objetos, paquetes, secuencia, colaboración, estados y actividades, implementación y despliegue) junto con su trazabilidad	<i>MagicDraw</i>
Análisis y Diseño	Se requiere de una herramienta de métricas del software que determine automáticamente el grado de bondad de los componentes de la arquitectura del software	<i>SonarQube</i>
Programación	Se requiere un entorno de desarrollo integrado para la edición, compilación, ejecución, ... del código en desarrollo en la máquina local	<i>Eclipse</i>

Disciplina	Necesidad	Herramientas
Programación	Se requiere ayudas para el cumplimiento de las reglas de estilo (formato, identificadores, ...) dadas en la arquitectura del software	<i>Eclipse, Checkstyle, PMD, FindBugs y Sonarqube</i>
Programación	Se requiere, en el contexto de metodologías ágiles, ayudas para automatizar en lo posible la refactorización del código (renombrado de identificadores, nombrar constantes, mover métodos, ...)	<i>Eclipse</i>
Programación	Se requiere un sistema de registro para gestionar (escritura, destino, avisos, ...) los mensajes de trazas, depuración, errores , ...) durante la ejecución	<i>Log4j</i>
Programación	Se requiere de un sistema de control de versiones del repositorio de código común del proyecto para facilitar la gestión (actualizaciones, vuelta atrás, mezclas, ...) de la rama de desarrollo, la rama de entregas, la rama de producción	<i>GitHub</i>
Pruebas	Se requiere un sistema para la gestión de pruebas que facilite la edición, ejecución, evaluación, ... de la pruebas	<i>Junit, Selenium, ...</i>
pruebas	Se requiere de un sistema de integración continua para comprobar que el código y las pruebas funcionan tras cualquier cambio	<i>Travis</i>
Pruebas	Se requiere un sistema de cobertura de pruebas que facilite la misión y estrategias de pruebas	<i>SonarQube</i>
Despliegue	Se requiere un gestor de proyectos para la automatización, en lo posible, de la construcción de entregables (compilación, pruebas, reglas de estilo, empaquetado, ...)	<i>Maven</i>
Gestión	De proyectos se requiere una herramienta para gestión de tickets que permitan la asignación de tareas con su tiempo estimado y real, finalización por parte del asignado y cierre tras la comprobación por el emisor de la tarea	<i>Tickets de GitHub</i>

Proceso de Desarrollo Software

- Mal llamado anteriormente como **Metodología**, cuando debió ser Método! y determina la disciplina de **Gestión**.
- La diferencia entre un Proceso de Desarrollo Software y otros radica en el **orden, grado y técnicas en que se acometen las actividades de las distintas Disciplinas de la Ingeniería del Software**
- **Proyecto Software**
 - Visión inicial, partido en dos fases de distinta naturaleza:
 - **Producción:** desde la toma de Requisitos a la primera entrega para explotación con los requisitos de partida
 - **Mantenimiento:** desde la primera entrega hasta la última para su explotación con los nuevos requisitos
 - Mantenimiento **correctivo**, para corregir defectos
 - Mantenimiento **perfectivo**, para modificar la funcionalidad
 - Mantenimiento **adaptativo**, para aceptar nuevas tecnologías, ...
 - Visión actual, un continuo de iteraciones de la misma naturaleza
 - **Desarrollo, Evolución:** desde la toma de Requisitos hasta la última entrega para su explotación con los nuevos requisitos



- *Analogía: el desarrollo software es como criar a un hijo; la producción del software sería la gestación del niño, que es un periodo vital en el crecimiento del hijo; el mantenimiento del software sería la educación del niño, que es un periodo largo y muy sorprendente que depende de la gestación y crianza previas*

Proceso de Desarrollo Software en Cascada

<ul style="list-style-type: none"> La versión original fue propuesta por Royce, W. W. en 1970 y posteriormente revisada por Boehm, B. en 1980 y Sommerville, I. en 1985. El inicio de cada etapa debe esperar a la finalización de la etapa anterior. Ante la detección de un error en una etapa se requiere la retroalimentación de fases anteriores 	
Desventajas	Ventajas

Desventajas	Ventajas
<ul style="list-style-type: none"> Son modelos fáciles de implementar y entender. Son modelos conocidos y utilizados con frecuencia. Promueven una metodología de trabajo efectiva: definir antes que diseñar, diseñar antes que codificar, ... Si el 90% o más de los requisitos de tu sistema se espera que sean estables a lo largo de la vida del proyecto, entonces aplicar una política dirigida por los requisitos es una oportunidad apropiada de dar razonablemente una solución óptima. [Booch – Object Solutions] 	<ul style="list-style-type: none"> Cualquier error de diseño detectado en la etapa de prueba conduce necesariamente al rediseño y nueva programación del código afectado, aumentando los costos del desarrollo Otros grados menores de estabilidad en los requisitos requieren un enfoque de desarrollo diferente para dar un valor tolerable del coste total [Booch – Object Solutions]

Proceso de Desarrollo Software Iterativo

- Iteración:** Un conjunto distinto de actividades llevado a cabo de acuerdo con un plan dedicado (iteración) y criterios de evaluación que se traduce en una entrega, ya sea interna o externa.
- Incremento:** Una parte del sistema pequeña y manejable, por lo general, la diferencia entre dos construcciones sucesivas.
 - Cada iteración se traducirá en al menos una nueva construcción y, por lo tanto, se añadirá un incremento en el sistema.

<p>“Planificar de un poco. Especificar, diseñar e implementar un poco. Integrar, probar y ejecutar un poco en cada iteración</p> <p style="text-align: right;">— Booch</p>	<ul style="list-style-type: none"> Los métodos o procesos de desarrollo software iterativos descomponen todas las actividades de cada disciplina en distintos momentos del proyecto que se repiten en mayor o menor medida, iteraciones. 	 Iteracion
---	--	---

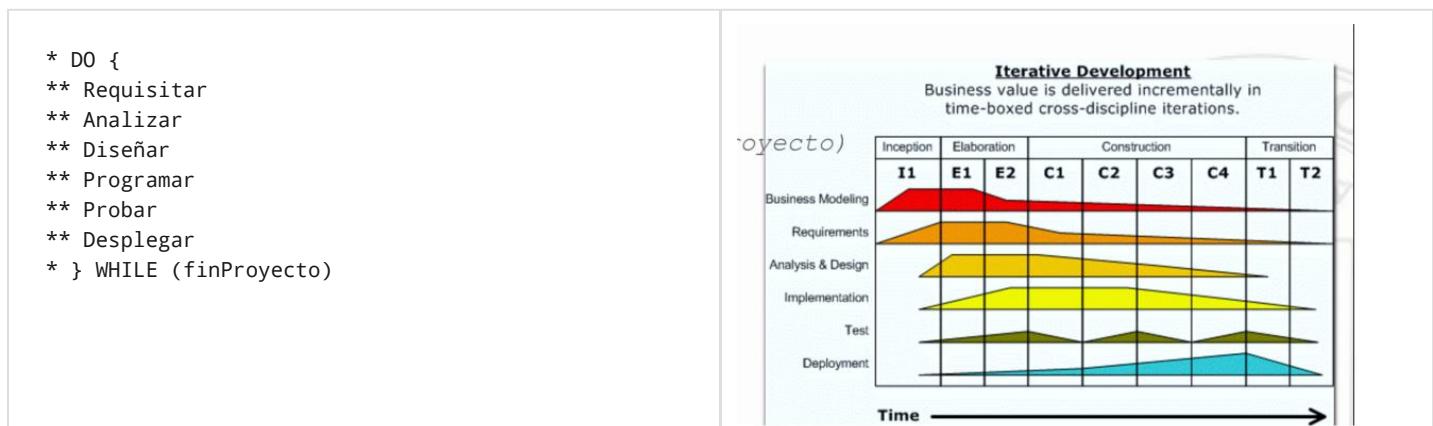
- Entrega.** Un conjunto de artefactos (documentos y posiblemente una construcción ejecutable) relativamente completo y consistente entregada a usuarios externos o internos
 - Entrega interna.** Una entrega no expuesta a los clientes y usuarios pero sí internamente solo para el proyecto y sus miembros
 - Entrega externa.** Una entrega expuesta a clientes y usuarios externos al proyecto y sus miembros.

Desventajas	Ventajas

Desventajas	Ventajas
<ul style="list-style-type: none"> Dificultad para gestionar a los miembros del equipo de desarrollo en un iteración cerrada o dificultad para gestionar a los miembros del equipo de desarrollo con varias iteraciones abiertas en paralelo 	<ul style="list-style-type: none"> Permite la participación del usuario desde fechas tempranas del proyecto para corregir las desviaciones de sus necesidades Permite elevar el ánimo del equipo de desarrollo con las entregas externas que superan las pruebas de aceptación Errores de programación, diseño, ... se localizan con facilidad en el incremento producido en la iteración vigente

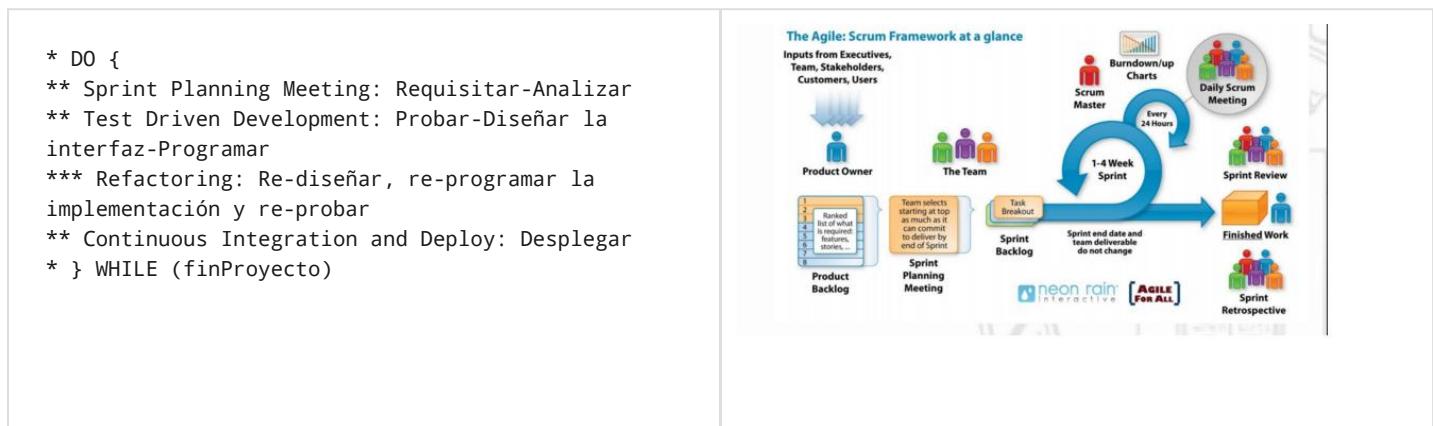
Proceso de Desarrollo Software Pesadas

- En 1998, *Grady Booch, Ivar Jacobson y Rumbaugh* unifican sus metodologías y publican *Rational Unified Process (RUP)*, un marco metodológico para grandes y pequeños proyectos



Proceso de Desarrollo Software Ágil

- En 2001, en Utah (EEUU), se reúnen un grupo de expertos, *Kent Beck* de eXtreme Programming (XP), *Martin Fowler*, *Robert Martin*, ... para defender un nuevo paradigma de desarrollo software:
<http://agilemanifesto.org/iso/es/manifesto.html>



Comparativa RUP vs XP

RUP	XP
Gestión de documentación de requisitos, análisis y diseño: pesadas/ligeras	Poca documentación porque la mejor documentación es el código: ligeras

RUP	XP
Con estimaciones del tiempo y coste del proyecto entero	Sin estimaciones del tiempo ni coste del proyecto entero
Planificación a largo plazo y revisiones del plan en cada iteración	Planificación de la iteración actual pero sin previsión a largo plazo
Entrevistas con el cliente durante las primeras iteraciones y en cada entrega	Entrevistas con el cliente durante todo el proyecto , cliente in situ parte del proyecto
Desarrollo priorizado por riesgos técnicos, políticos, ...	Desarrollo priorizado por el valor de retorno al cliente
Roles especializados por desarrollador	Desarrolladores multidisciplinares
Diseño de la Arquitectura propuesta previa al desarrollo	Arquitectura emergente según se re-diseña (TDD y refactoring)

Diseño

Santa Tecla
parqueNaturalSantaTecla@gmail.com
Version 0.0.1

Índice

Justificación: ¿Por qué?

Definición: ¿Qué?

Objetivos: ¿Para qué?

Descripción: ¿Cómo?

Modelo del Dominio

 Estrategias de Clasificación

 Relaciones entre Clases

 Antipatrón “Descomposición Funcional”

Legibilidad

 Formato

 Comentarios

 Nombrado

 YAGNI

 Estándares

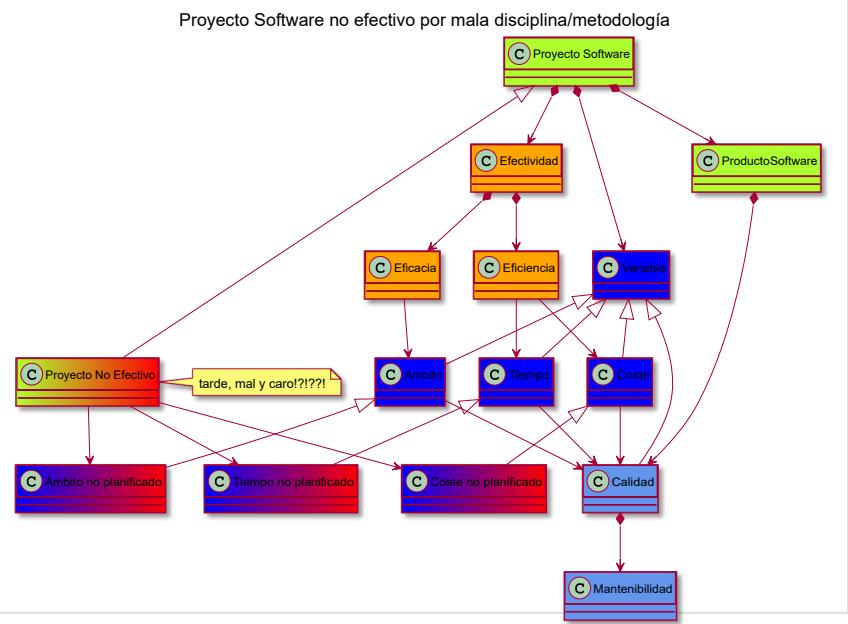
 Consistencia

 Alertas

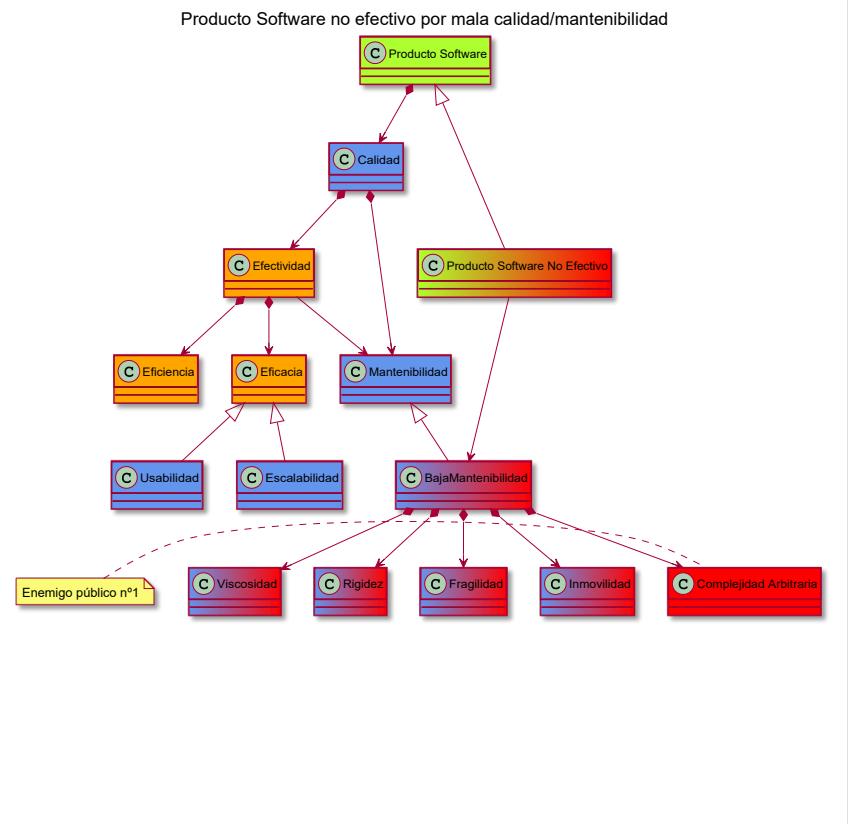
Bibliografía

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
 - **inmóvil**, porque no se puede reutilizar con facilidad

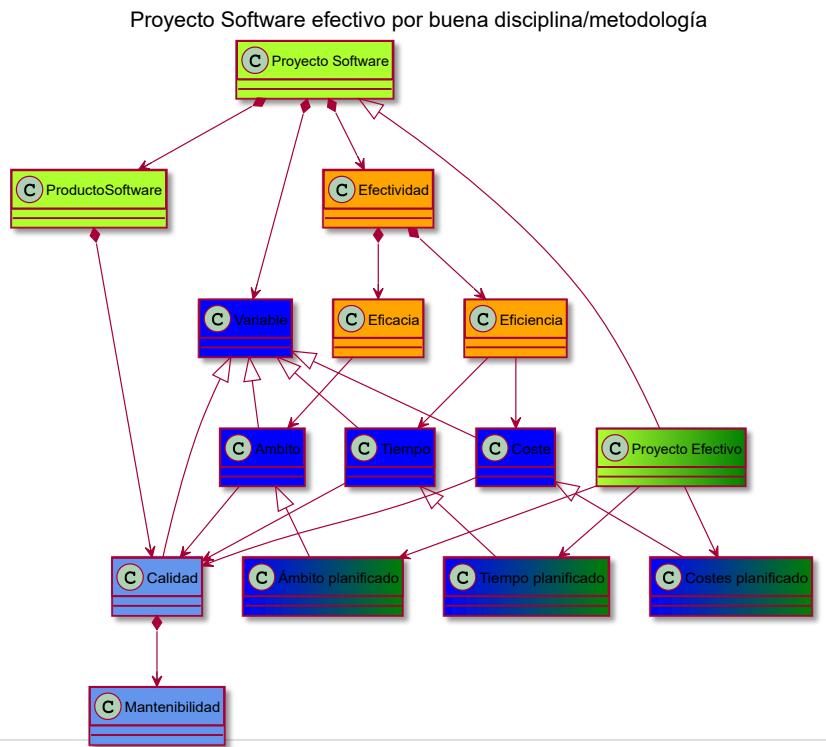


Definición: ¿Qué?

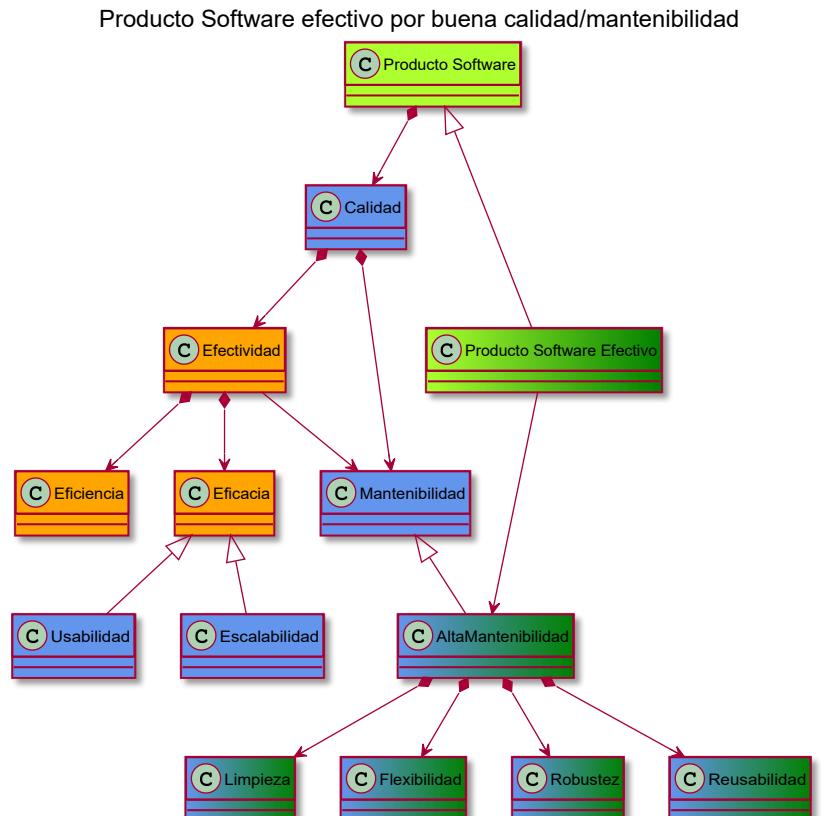
- La disciplina de diseño es el flujo de trabajo, incluyendo actividades, trabajadores y documentos, cuyo principal propósito es **desarrollar enfocados en los requisitos no funcionales y en el dominio de la solución para preparar para la implementación y pruebas del sistema:**
 - Adquirir una comprensión profunda sobre los aspectos de los requisitos no funcionales y limitaciones relacionadas con:
 - los lenguajes de programación,
 - la reutilización de componentes,
 - sistemas operativos,
 - tecnologías de distribución y concurrencia,
 - tecnologías de bases de datos,
 - tecnologías de interfaz de usuario,
 - tecnologías de gestión de transacciones,
 - ...

Objetivos: ¿Para qué?

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



- Producto Software efectivo
 - porque tiene **buenas calidad**
 - Es eficiente
 - Es eficaz en **corrección, usabilidad, escalabilidad, ...**
 - porque tiene **buenas 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
 - **fuerte**, porque sí se puede probar con facilidad
 - **reusable**, porque sí se puede reutilizar con facilidad



• Objetivos

- Crear una entrada apropiada como **punto de partida para las disciplinas posteriores** mediante la captura de los requisitos correspondientes a los distintos subsistemas, interfaces y clases
- Capacitar para la **descomposición del trabajo** de implementación en piezas más manejables gestionados por diferentes equipos de desarrollo, posiblemente al mismo tiempo

- Captura las **interfaces principales entre los subsistemas** del ciclo de vida del software. Esto es útil cuando razonamos sobre la arquitectura y cuando usamos las interfaces como instrumentos de sincronización entre los diferentes equipos de desarrollo
- Capacitar para **visualizar y razonar sobre el diseño** utilizando una notación común
- Crear una abstracción sin fisuras de la implementación del sistema, en el sentido de que la aplicación es un refinamiento sencillo del diseño mediante la cumplimentación de la "carne", pero sin cambiar la estructura, el esqueleto. Esto permite el uso de técnicas como la generación de código con **ingeniería directa e inversa** entre el diseño y la implementación

Descripción: ¿Cómo?

Modelo del Dominio

- **Patrón de Experto en la información:**
 - **Principio general** de la asignación de responsabilidades a los objetos, aplicación directa del modelo del dominio
 - **Asignar la responsabilidad a la clase que tiene la información necesaria para cumplir con la responsabilidad**
 - Tenga en cuenta que el cumplimiento de una responsabilidad a menudo requiere la información que se transmite a través de diferentes clases de objetos. Esto implica que hay **muchos expertos "parciales" de información** que colaborarán en la tarea.
 - Tiene una **analogía en el mundo real**, como muchas cosas en la tecnología de objetos. Conduce a diseños en los que un objeto de software hace las **operaciones que realizan normalmente las cosas del mundo real inanimadas** a las que representa.

“Cuando los programadores piensan en los problemas, en términos de comportamientos y responsabilidades de los objetos, traen con ellos un caudal de intuición, ideas y conocimientos provenientes de su experiencia diaria”

— Budd

Programación Orientada a Objetos. 1994

“En lugar de un saqueador de bits que saquea estructuras de datos, nosotros tenemos un universo de objetos con buen comportamiento que cortésmente se solicitan entre sí cumplir diversos deseos”

— Ingalls

Design Principles Behind Smalltalk. Byte vol. 6(8)

Estrategias de Clasificación

- **Experto en la información:**
 - **Principio general** de la asignación de responsabilidades a los objetos, aplicación directa del modelo del dominio
 - **Asignar la responsabilidad a la clase que tiene la información necesaria para cumplir con la responsabilidad**
 - Tenga en cuenta que el cumplimiento de una responsabilidad a menudo requiere la información que se transmite a través de diferentes clases de objetos. Esto implica que hay **muchos expertos "parciales" de información** que colaborarán en la tarea.
 - Tiene una **analogía en el mundo real**, como muchas cosas en la tecnología de objetos. Conduce a diseños en los que un objeto de software hace las **operaciones que realizan normalmente las cosas del mundo real inanimadas** a las que representa.

Descripción informal

- Abbott sugiere escribir una descripción del problema (o una parte de un problema) y luego subrayar los sustantivos y verbos. Los **nombres representan objetos candidatos**, y los **verbos representan operaciones candidatos** en ellos. El enfoque de Abbott es útil porque es simple y porque obliga a los desarrolladores a trabajar en el vocabulario del espacio del problema.
- **Inconveniente:**
 - Sin embargo, de ninguna manera es un enfoque riguroso y sin duda **no escala bien** para nada más allá de problemas bastante triviales. El lenguaje humano es un vehículo de expresión tremadamente impreciso, por lo que la calidad de la lista resultante de los objetos y las operaciones depende de la habilidad de la escritura de su autor: **anáforas, metáforas, ...**

- Por otra parte, cualquier sustantivo puede ser verbo, y cualquier verbo puede ser sustantivo, **cosificación**. Ej.: gestionar vs gestión; oxígeno vs oxigenar;

Análisis clásico

- Un número de metodólogos han propuesto **diversas fuentes de clases y objetos**, derivados de los requisitos del dominio del problema:
 - **Cosas, objetos físicos o grupos de objetos que son tangibles**: coches, datos de telemetría, sensores de presión, ...
 - **Conceptos, principios o ideas no tangibles** que se utilizan para organizar o realizar un seguimiento de las actividades comerciales y/o comunicaciones: préstamo, reunión, intersección
 - **Cosas que pasan**, por lo general de otra cosa en una fecha determinada, eventos: aterrizaje, interrumpir, solicitud
 - **Gente, seres humanos** que llevan a cabo alguna función, usuarios que juegan diferentes roles en la interacción con la aplicación: madre, profesor, político
 - **Organizaciones, colecciones formalmente organizadas de personas y recursos** que tienen una misión definida, cuya existencia es en gran medida independiente de los individuos
 - **Lugares físicos, oficinas y sitios** importantes para la aplicación: zonas reservadas para personas o cosas
 - **Dispositivos** con los que interactúa la aplicación
 - **Otros sistemas de sistemas externos** con los que interactúa la aplicación

Análisis del Dominio

- Un intento para identificar los **objetos, las operaciones y las relaciones** [son los que] los expertos de dominio **perciben** como importantes sobre el dominio.
 - A menudo, un experto de dominio es **simplemente un usuario**, como un ingeniero del tren o expendedor en un sistema ferroviario, o una enfermera o un médico en un hospital.
 - Un experto del dominio normalmente no será un desarrollador de software; más comúnmente, él o ella es simplemente una **persona que está íntimamente familiarizado con todos los elementos de un problema** particular.
 - Un experto del dominio habla el **vocabulario del dominio problema**.

Análisis del Comportamiento

- Mientras que estos enfoques clásicos se centran en cosas tangibles en el dominio del problema, otra escuela de pensamiento en el análisis orientado a objetos se **centra en el comportamiento dinámico** como la fuente primaria de clases y objetos.
 - En esta estrategia hacen hincapié en las responsabilidades, que denotan "**el conocimiento de un objeto mantiene y las acciones que un objeto puede realizar**". Las responsabilidades tienen el propósito de transmitir un sentido de la finalidad de un objeto y su lugar en el sistema. Las responsabilidades de un objeto son todos los servicios que presta a todos los contratos que apoya_“ [Wirfs-Brock, Wilkerson y Wiener]
- **Responsabilidades**: las obligaciones de un objeto en términos de su comportamiento. Existen dos tipos básicos:
 - La **responsabilidad de hacer de un objeto** es: algo en sí mismo, como la creación de un objeto o hacer un cálculo, iniciar acciones en otros objetos y el control y la coordinación de actividades en otros objetos
 - La **responsabilidad de conocer de un objeto** es: sobre unos datos privados encapsulados, sobre objetos relacionados, y sobre las cosas que pueden obtener o calcular
 - Se implementan utilizando métodos que, o bien actúan solos o colaboran con otros métodos y objetos. Una responsabilidad no es lo mismo que un método y se ve influida por la granularidad de la responsabilidad.

- Por ejemplo:
 - El acceso a las bases de datos relacionales puede implicar decenas de clases y cientos de métodos, empaquetados en un subsistema.
 - Por el contrario, la responsabilidad de "crear una venta" puede implicar sólo uno o unos métodos
- Estrategia:
 - A medida que los **miembros del equipo caminan a través del escenario**, pueden asignar **nuevas responsabilidades** a una clase existente, **agrupar ciertas responsabilidades** para formar una nueva clase, o más comúnmente, **dividen las responsabilidades** de una clase en más de grano fino y **distribuyen estas responsabilidades** a clases diferentes.
 - Se crea una tarjeta para cada clase identificada como relevantes para el escenario: **Tarjetas CRC (class-responsability-colaborations)** han demostrado ser una herramienta de desarrollo útil que facilita el intercambio de ideas y mejora la comunicación entre los desarrolladores. Una tarjeta CRC no es más que una tarjeta de 3x5 pulgadas (7x12,5 cms), en la que el analista escribe en lápiz el nombre de una clase (en la parte superior de la tarjeta), sus responsabilidades (en un medio de la tarjeta), y sus colaboradores (en la otra mitad de la tarjeta).
 - Hoy en día está subsumido por las herramientas CASE con UML: la clase y la responsabilidad de cada Tarjeta CRC está en los nodos del grafo que forman las clases del diagrama y las colaboraciones, hoy dependencias, están en los arcos del grafo

Head	
Responsibilities:	Collaborators:
- contains a face and eyes - updates the appearance of its lips	- Face, EyeMorph - LipMorph
AnimationEvent	
Responsibilities	Collaborators
- has a start time	
FaceEvent (AnimationEvent)	
Responsibilities:	Collaborators:
- instructs the face to animate (blink, frown, smile, etc)	- FaceMorph
NodEvent (AnimationEvent)	
Responsibilities:	Collaborators:
- instructs the head to perform a nodding action	- Head
TalkEvent (AnimationEvent)	
Responsibilities:	Collaborators:
- instructs the head to begin speaking	- Head, SqueakSpeaker
SqueakSpeaker	
Responsibilities:	Collaborators:
- creates a voice	- Actor

Análisis de Casos de Uso

- A medida que el equipo se guía **a través de cada escenario de cada caso de uso**, se deben identificar los objetos que participan en el escenario, las **responsabilidades de cada objeto**, y las formas en esos **objetos colaboran con otros objetos**, en términos de las operaciones de cada uno invoca en el otro. De esta manera, el equipo se ve obligado a elaborar una clara separación de las responsabilidades entre todas las abstracciones.
- **No es necesario profundizar** en estos escenarios al principio; simplemente podemos enumerarlos. Estos escenarios describen colectivamente las funciones del sistema de la aplicación.
- A medida que continúa el proceso de desarrollo, estos **escenarios iniciales se ampliaron para considerar las condiciones excepcionales**, así como los comportamientos secundarios del sistema. Los resultados de estos escenarios secundarios introducen nuevas abstracciones para añadir, modificar o reasignar las responsabilidades de abstracciones existentes.

- Entonces se procede por un estudio de cada escenario, posiblemente utilizando técnicas similares a las **prácticas de la industria de la televisión: storyboard** (guión gráfico) y películas.
- Los escenarios también sirven como la base de las **pruebas del sistema**.

Relaciones entre Clases

“Un objeto en si mismo no es interesante. Los objetos contribuyen al comportamiento de un sistema colaborando con otros objetos”

— Grady Booch
Análisis y Diseño Orientado a Objetos. 1996

- Dependencia** es la nueva forma de referirse a una relación entre clases. **Tipos**
 - Por **colaboración**, si dos objetos colaboran sus respectivas clases están relacionadas. Tipos de relación:
 - Relación de Composición/Agregación**
 - Relación de Asociación**
 - Relación de Dependencia (Uso)**
 - Por **transmisión**, si una clase transmite a otra todos sus miembros para organizar una jerarquía de clasificación, sin negar ni obligar a la colaboración entre objetos de las clases participantes. Tipos de relación:
 - Relación de Herencia por Extensión**
 - Relación de Herencia por Implementación**

Características

Característica	Definición	Ejemplos
Visibilidad	carácter privado o público de la colaboración entre dos objetos, o sea si otros objetos o no colaboran también con el que recibe los mensajes.	un profesor colabora con de forma privada con su bolígrafo que mordisquea y nadie más “colabora” con él y colabora con un proyector del aula y otros profesores también colaboran con él
Temporalidad	mayor o menor duración de la colaboración entre dos objetos que colaboran.	un profesor colabora un tiempo “reducido” (5 horas!) con el proyector del aula y, por tiempo “indefinido” colabora con su computadora con todos sus documentos, instalaciones, ...
Versatilidad	intercambiabilidad de los objetos en la colaboración con otro objeto.	un profesor colabora con su computadora para preparar la documentación de un curso y colabora con cualquier computadora para consultar el correo electrónico

Relación de Composición/Agregación

- Es la relación que se constituye entre **el todo y la parte**. Se puede determinar que existe una relación de **composición entre la clase A, el todo, y la clase B, la parte, si un objeto de la clase A “tiene un” objeto de la clase B**.
 - La relación de composición no abarca simplemente **cuestiones físicas** (*libro -todo- y páginas -parte-*) sino, también, a **relaciones lógicas** que respondan adecuadamente al todo y a la parte como “contiene un” (*aparato digestivo -todo- y bolo alimenticio -parte-*), “posee un” (*propietario -todo- y propiedades -parte-*), etc.
 - Las **características** de la relación de composición/agregación son:

- visibilidad privada y pública respectivamente
- temporalidad no momentánea
- versatilidad es frecuentemente reducida
- La diferencia entre composición y agregación:
 - La **composición** es una **forma fuerte de composición donde la vida de la clase contenida debe coincidir con la vida de la clase contenedora**. Los componentes constituyen una parte del objeto compuesto. De esta forma, los componentes **no pueden ser compartidos** por varios objetos compuestos. La supresión del objeto compuesto conlleva la supresión de los componentes.
 - *Por ejemplo: persona y cabeza; una cabeza solo puede pertenecer a una persona y no puede existir una cabeza sin su persona*
 - La **agregación** es un tipo de asociación que indica que una clase es parte de otra clase (**composición débil**). Los componentes **pueden ser compartidos** por varios compuestos. La **destrucción del compuesto no conlleva la destrucción de los componentes**.
 - *Por ejemplo: persona y familia; un persona puede pertenecer a la familia en que nació y a la que posteriormente formó y seguir vivo aunque ya no existan dichas familias*

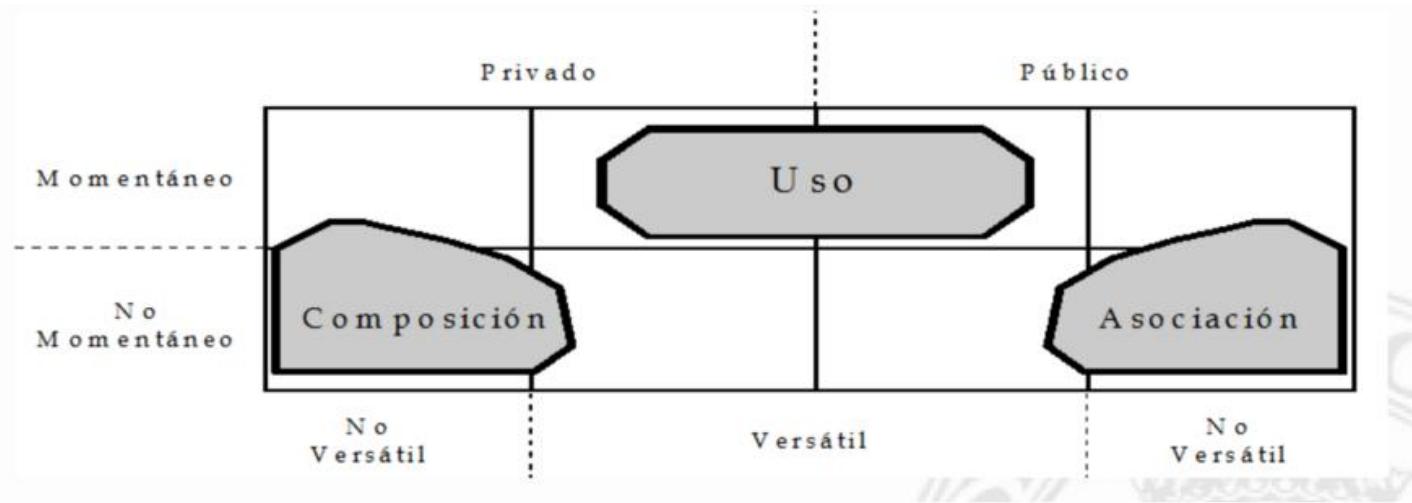
Relación de Asociación

- Es la relación que perdura entre un cliente y un servidor determinado. Existe una relación de **asociación entre la clase A, el cliente, y la clase B, el servidor, si un objeto de la clase A disfruta de los servicios de un objeto determinado de la clase B (mensajes lanzados) para llevar a cabo la responsabilidad del objeto de la clase A en diversos momentos** creándose una dependencia del objeto servidor.
 - La relación de asociación no abarca simplemente **cuestiones tangibles** (procesador -cliente- y memoria - servidor-) sino, también a **cuestiones lógicas** que respondan adecuadamente al cliente y al servidor determinado como “provecho” (socio -cliente- y club -servidor-), “beneficio” (empresa -cliente- y banca - servidor-), etc.
 - Las **características** de la relación de asociación son:
 - visibilidad pública
 - temporalidad no momentánea
 - versatilidad es frecuentemente reducida

Relación de Dependencia (uso)

- Es la relación que se establece **momentáneamente entre un cliente y cualquier servidor**. Existe una **relación de uso entre la clase A, el cliente, y la clase B, el servidor, si un objeto de la clase A disfruta de los servicios de un objeto de la clase B (mensajes lanzados) para llevar a cabo la responsabilidad del objeto de la clase A en un momento** dado sin dependencias futuras.
 - La relación de uso no abarca simplemente **cuestiones tangibles** (ciudadano -cliente- y autobús -servidor-) sino, también a **cuestiones lógicas** que respondan adecuadamente al cliente y al servidor momentáneo cualquiera que sea como “goce” (espectador -cliente- y actor -servidor-), “beneficio” (viajante -cliente- y motel -servidor-), etc.
 - Las **características** de la relación de dependencia (uso) son:
 - visibilidad pública o privada
 - temporalidad momentánea
 - versatilidad es alta

Comparativa entre Relaciones



- Sin duda, falta mencionar el factor más determinante a la hora de decidir la relación entre las clases: el **contexto** en el que se desenvuelvan los objetos. Éste **determinará de forma “categórica” qué grados de visibilidad, temporalidad y versatilidad se producen en su colaboración**. Por ejemplo:
 - Si el contexto de los objetos paciente y médico es un hospital de urgencias la relación se decantaría por un uso mientras que si es el médico de cabecera que conoce su historial y tiene pendiente algún tratamiento, la relación se inclinaría a una asociación;
 - Si el contexto de los objetos motor y coche es un taller mecánico (se accede al motor de un coche, se cambian motores a los coches, etc.) la relación se inclinaría a una asociación, mientras que si el contexto es la gestión municipal del parque automovilístico (se da de alta y de baja al coche, se denuncia al coche, etc. y el motor se responsabiliza de ciertas características que dependen del ministerio de industria como su potencia fiscal, etc.) la relación se inclinaría a una composición

“La decisión de utilizar una agregación es discutible y suele ser arbitraria. Con frecuencia, no resulta evidente que una asociación deba ser modelada en forma de agregación. En gran parte, este tipo de incertidumbre es típico del modelado; este requiere un juicio bien formado y hay pocas reglas inamovibles. La experiencia demuestra que si uno piensa cuidadosamente e intenta ser congruente la distinción imprecisa entre asociación ordinaria y agregación no da lugar a problemas en la práctica.

— Rumbaugh
1991

- **No existe para toda colaboración un relación ideal categórica.** Es muy frecuente que sean varias relaciones candidatas, cada una con sus ventajas y desventajas. Por tanto, al existir diversas alternativas, será una decisión de ingeniería, un compromiso entre múltiples factores no cuantificables: costes, modularidad, legibilidad, eficiencia, etc., la que determine la relación final.

Antipatrón “Descomposición Funcional”

Sinónimos	Synonyms	Libro	Autor
Descomposición Funcional	Functional Descomposition	Antipatrón de Desarrollo	

- **Síntomas:** clases con nombres de función, clases con un solo método, Ausencia de principios orientados a objetos como herencia, polimorfismo, abuso de miembros estáticos, ...
- **Justificación:** Imposible de comprender el software, de reutilizar, de probar, ...
- **Solución:** Aplicar las pautas del Modelo del Dominio Orientado a Objetos
- Ejemplo TicTacToe (<https://github.com/miw-upm/TWVG/tree/master/doo/src/main/java/ticTacToe/v040>)

Legibilidad

“Una línea de código se escribe una vez y se lee cientos de veces”

— Tom Love

Formato

- **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 permanecer juntas
 - Las variables deberían declararse tan cerca como sea posible de su utilización, **minimizar el intervalo de vida de una variable**
 - 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 declarados 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.

- **Violaciones:**

- No uses **tabuladores** entre los tipos y las variables para una disposición por columnas
- **Nunca rompas las reglas de sangrado** por muy pequeñas que sean las líneas

“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.”

— Martin R.

- Ejemplo TicTacToe (<https://github.com/miw-upm/IWVG/tree/master/doo/src/main/java/ticTacToe/v010>)

Comentarios

- **Justificación:** Nada puede ser tan útil como un comentario bien colocado.

- **Implicaciones:**

- Comentario **legal**. Ej.: copyright, license, ...

- Comentario **aclarativo**. Ej.: `//format matched kk:mm:ssEEE, MMM dd, yyyy; String format = "||d*:||d*:||d* ||w*, ||w* ||d*; ||d*";`
- 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.

- **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 javadoca 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|||||||||||||||||||||||||`
- Comentarios **no mantenidos**. Con valores que no se actualizará. Ej.: `//portis7077`
- 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
- **Código comentado**. Para eso está el control de versiones

“ “No comentes código malo, reescribelo”

— Kernighan & Plaugher

- Ejemplo TicTacToe (<https://github.com/miw-upm/IWVG/tree/master/doo/src/main/java/ticTacToe/v020>)

Nombrado

Sinónimos	Synonyms	Libro	Autor
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 name	Clean Code	Robert Martin
Usa nombres largo para ámbitos largos	Use long names for long scopes	Clean Code	Robert Martin

Sinónimos	Synonyms	Libro	Autor
Evita codificaciones	Avoid Encodings	Clean Code	Robert Martin
Los nombre debería describir los efectos laterales	The names should describe the side effects	Clean Code	Robert Martin

- **Justificación:** Los nombre deben **revelar su intención**. Deberían revelar por qué existe, qué hace, y cómo se utilizapara facilitar la legibilidad para el desarrollo y el mantenimiento correctivo, perfectivo y adaptativo
 - 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 el miedo y consideramos 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.
- **Implicaciones:**
 - Nombres **pronunciables** que permitan mantener una conversación
 - **Mayúsculas en los cambios de palabra** (*CamelCase*). Ej.: *FireWeaponController*
 - 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.
 - Nombres de **paquetes** deben ser sustantivosy 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 o put*
- **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
 - **Constantes numéricas** que son difíciles de localizar y mantener
 - Nombres de **una letra** y muy en particular, 'O' y 'I' que se confunden con 0 y 1. Excepcionalmente, en variables locales auxiliares de métodos. Un contador de bucle puede ser nombrado *io* *jo* *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 *Behaviour Human Prediction System*
 - Nombres con **códigos de tipo o información del ámbito** (notación Húngara y similares). Ej.: *int iAgeo intm_iAge*, ... mejor *age*; *class CStudent*, mejor *Student*
 - Nombre con **palabras vacías de significado** 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 `XYZControllerForEfficientHandlingOfStrings` y `XYZControllerForEfficientStorageOfStrings`
- Nombres **polisémicos** en un mismo contexto. Ej.: `book` como registro en un hotel y libro; ...
- Ejemplo TicTacToe (<https://github.com/miw-upm/IWVG/tree/master/doo/src/main/java/ticTacToe/v030>)

YAGNI

Sinónimos	Synonyms	Libro	Autor
No lo vas a necesitar o no lo vas a necesitar	YAGNI You aren't going to need it o You ain'tgonnaneed it		
Generalidad Especulativa	Speculative Generality	Smell Code(Refactoring)	Martin Fowler

• Código sucio:

- 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

• Violaciones:

- 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.
- Ejemplo TicTacToe (<https://github.com/miw-upm/IWVG/tree/master/doo/src/main/java/ticTacToe/v110>): *Clase IO*

Estándares

Sinónimos	Synonyms	Libro	Autor
Seguir las convención estándar	Follow Standard Conventions	Clean Code	Robert Martin

- Siga las convenciones **estándar basadas en normas comunes de la industria** [*Clean Code - Robert Martin*]
 - Debe especificar cosas como dónde declarar variables de instancia; cómo nombrar las clases, métodos y variables; dónde poner los paréntesis, las llaves; ...
- 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.

- **No se necesita un documento** para describir estos convenios porque su código proporciona los ejemplos.

Consistencia

Sinónimos	Synonyms	Libro	Autor
Inconsecuencia	Inconsistency	Clean Code	Robert Martin

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

Alertas

Sinónimos	Synonyms	Libro	Autor
Seguridad anulada	Overridden Safeties	Clean Code	Robert Martin

- Es **arriesgado desactivar ciertas advertencias** del compilador (o todas las advertencias!) aunque puede ayudarle a obtener la sensación de tener éxito pero con el riesgo de sesiones de depuración sin fin.
 - *Por ejemplo: 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.*

Código Muerto

Sinónimos	Synonyms	Libro	Autor
Antipatrón	Dead Code		
Flujo de lava	Lava Flow		

- **Síntomas:**
 - Fragmentos de código injustificables, inexplicables u obsoletas en el sistema: interfaces, clases, funciones o segmentos de código complejo con aspecto importante pero que **no están relacionados con el sistema**
 - Bloques de **código comentado** sin explicación o documentación
 - Bloques de **código con comentarios** //TODO: “proceso de cambio”, “para ser reemplazado”, ...
- **Antipatrón:**
 - Segundo 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.
- Ejemplo TicTacToe (<https://github.com/miw-upm/IWVG/tree/master/doo/src/main/java/ticTacToe/v100>): *Método clear de Clase Board*

DRY

Sinónimos	Sinónimos	Libro	Autor
No te repitas	DRY(Don't Repeat Yourself)		
Fuente Única de la Verdad	Single Source of Truth		

Antónimos	Antonyms	Libro	Autor
Código Duplicado	Duplicate code)	Smell Code -(Refactoring)	Martin Fowler
Copiar y Pegar	Copy+Paste	Antipatrón de Desarrollo	William 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		

• Justificación:

- **Evitar re-analizar, re-diseñar, re-codificar, re-probar y re-documentar** soluciones que complican enormemente el mantenimiento correctivo, perfectivo y adaptativo
 - *Por ejemplo: 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 1^aFN de BBDD).
- **Aplicable a todo:** la programación, esquemas de bases de datos, planes de prueba, el sistema de construcción, análisis y diseños, incluso la documentación.

• 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.
 - *Por ejemplo: 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**
- Ejemplo TicTacToe (<https://github.com/miw-upm/IWVG/tree/master/doo/src/main/java/ticTacToe/v120>): *Método de Clases Board y Player*

Diseño Modular

Santa Tecla
parqueNaturalSantaTecla@gmail.com
Version 0.0.1

Índice

Justificación: ¿Por qué?

Definición: ¿Qué?

Objetivos: ¿Para qué?

Descripción: ¿Cómo?

Divide y Vencerás

Número de módulos

Distribución de Responsabilidades

Diseño dirigido por Niveles

Interfaz

Interfaz Suficiente, Completa y Primitiva

Principios del Menor Compromiso y la Menor Sorpresa

Código Sucio por Clases Alternativas con Interfaces Diferentes

Diseño por Contrato

Implementación

Cohesión

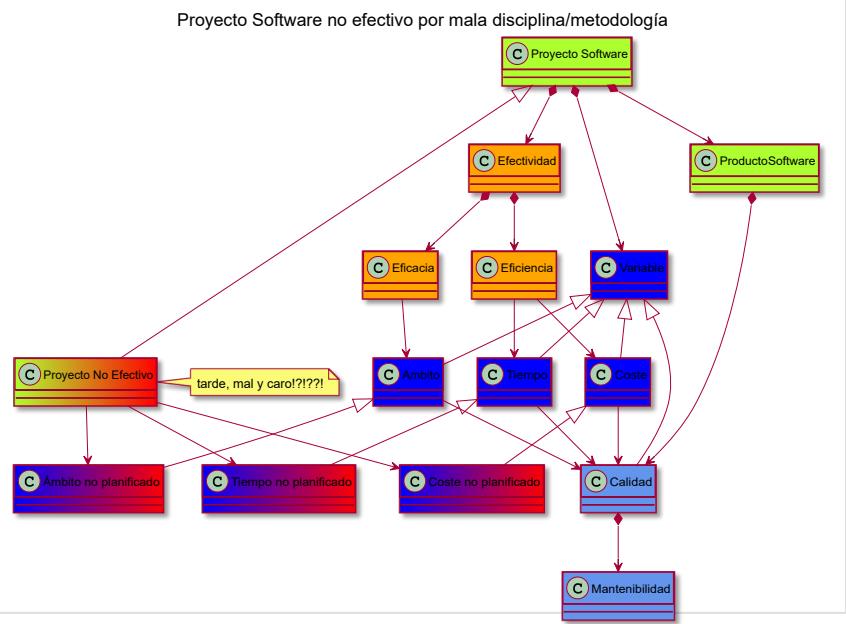
Acoplamiento

Tamaño

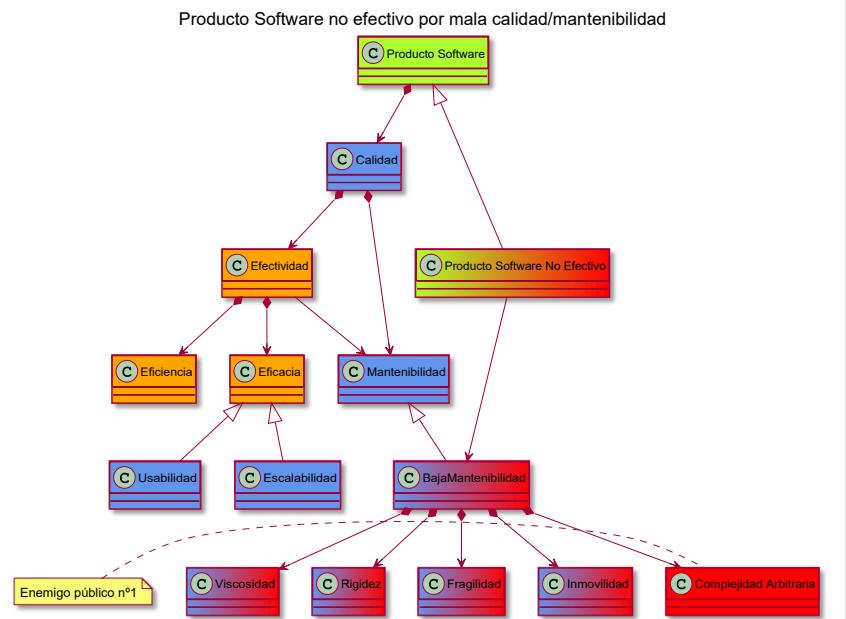
Bibliografía

Justificación: ¿Por qué?

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

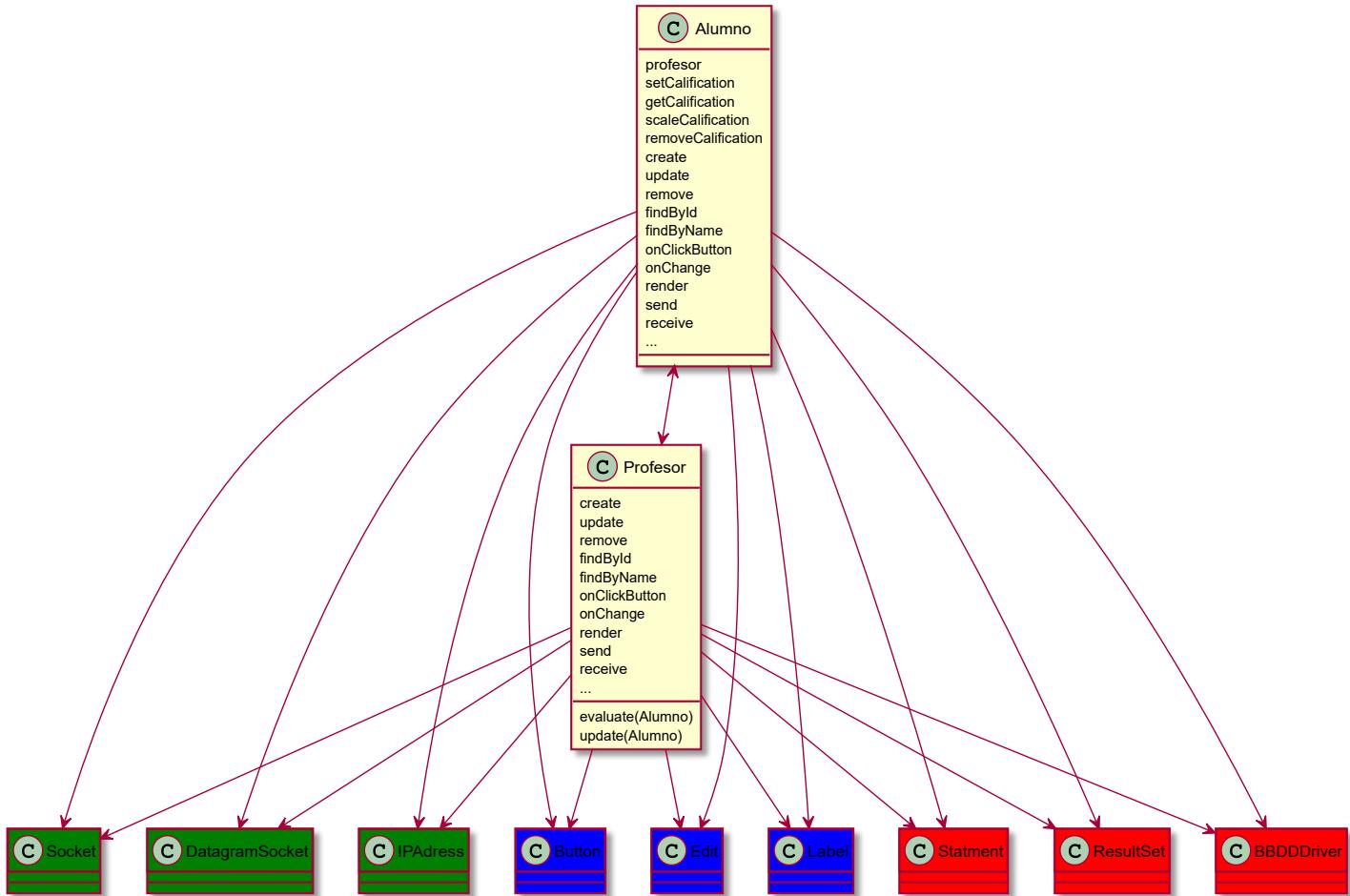


- Producto Software poco efectivo
 - porque tiene **mala calidad**
 - **Poco eficiente**
 - Poco eficaz en **corrección, usabilidad, escalabilidad, ...**
 - porque tiene **mala mantenibilidad**, de la que depende la eficiencia y la eficacia anteriores, porque es
 - **viscoso**, porque no se puede entender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



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

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



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

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

Definición: ¿Qué?



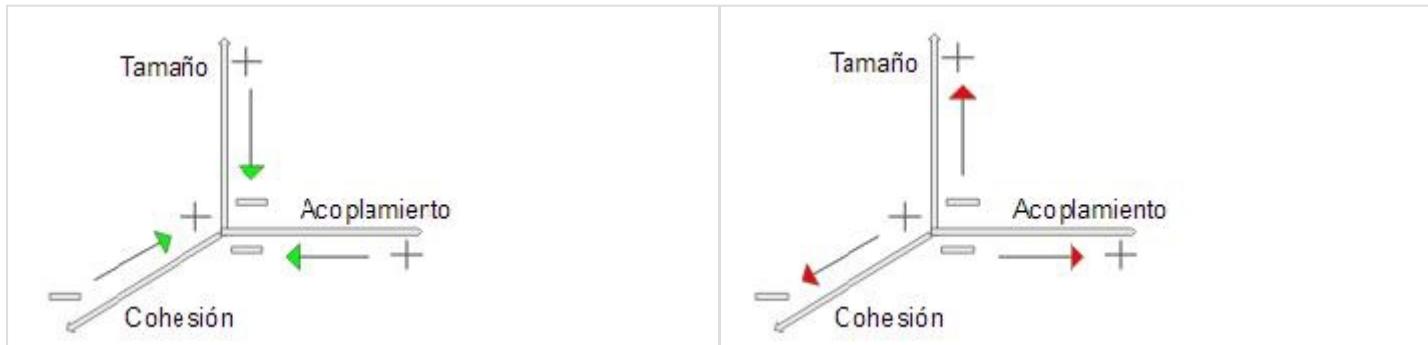
- Se basa en:

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

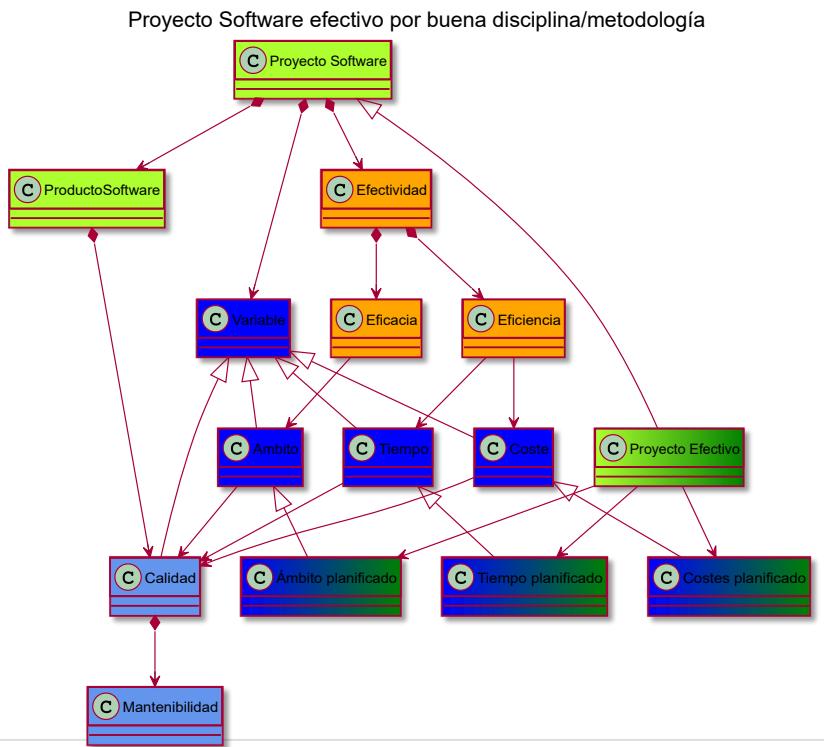
Alta cohesión	<ul style="list-style-type: none"> • Única responsabilidad, motivo de cambio, como máximo para cada método, clase y paquete
Bajo acoplamiento	<ul style="list-style-type: none"> • paquetes con 5 paquetes dependientes máximio • clases con 5 clases dependientes máximo

Tamaño pequeño	<ul style="list-style-type: none">• paquetes con 20 clases máximo• clases con 5 atributos máximo• clases con 20 métodos máximo• métodos con 2 parámetros máximo• métodos con una media de 1,2 parámetros• métodos con 25 líneas como máximo• métodos con 3 niveles de anidación como máximo• métodos con una complejidad algorítmica, número de caminos, de 12 como máximo
-----------------------	---

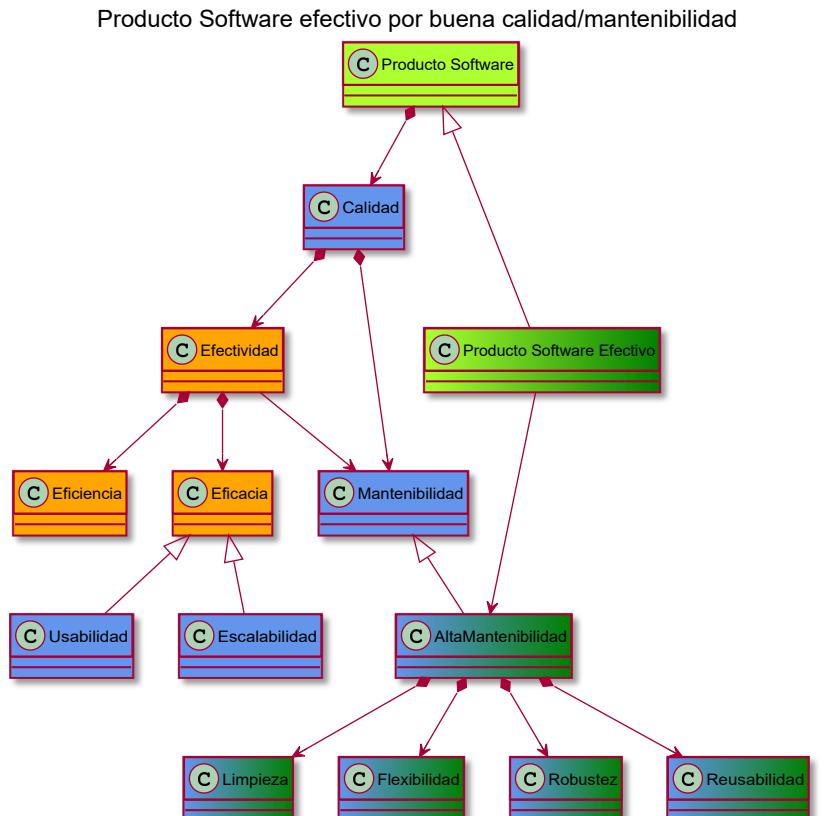


Objetivos: ¿Para qué?

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



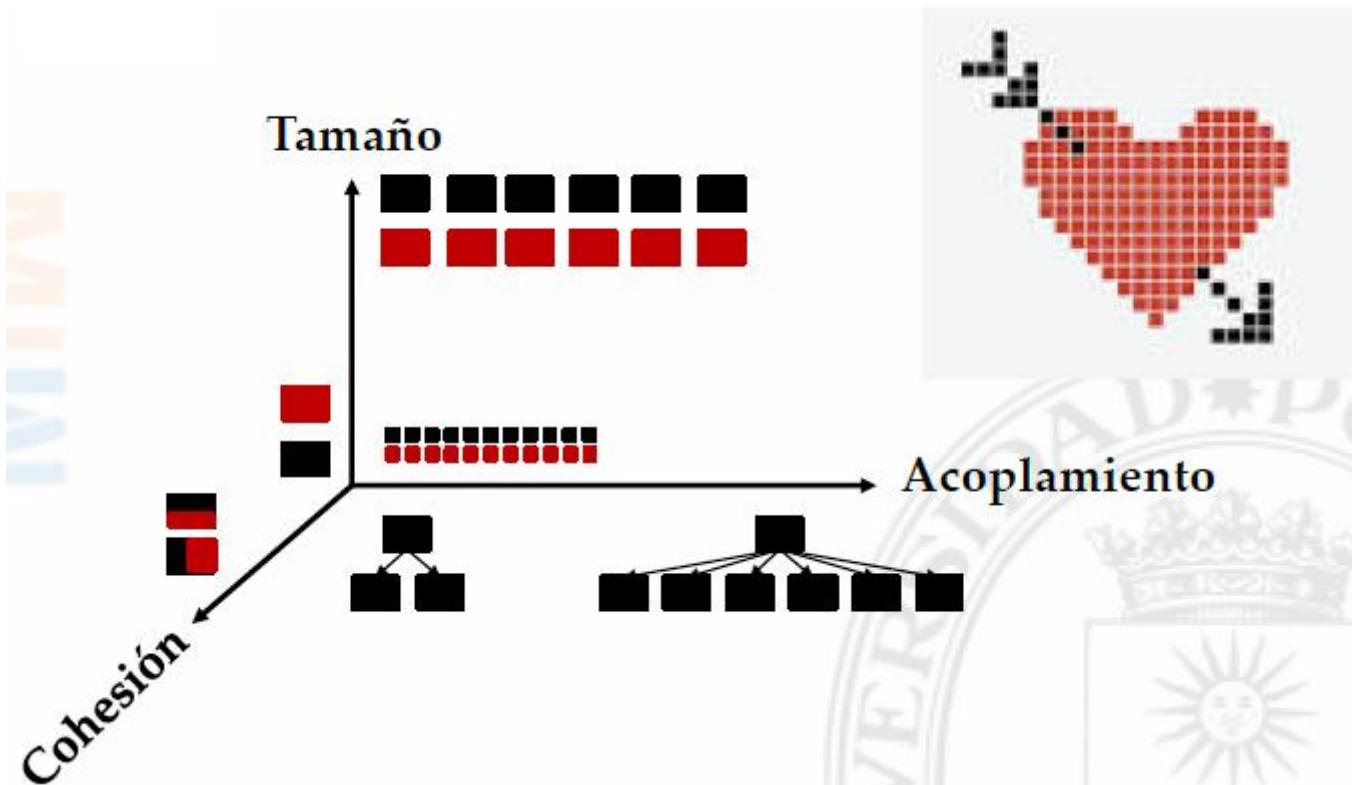
- Producto Software efectivo
 - porque tiene **buenas calidad**
 - Es eficiente
 - Es eficaz en **corrección, usabilidad, escalabilidad, ...**
 - porque tiene **buenas mantenibilidad**, de la que depende la eficiencia y la eficacia anteriores, porque es
 - **ligero**, porque sí se puede entender con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



Fluido

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

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



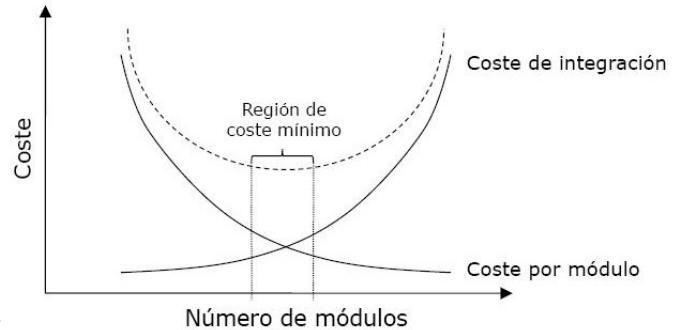
Descripción: ¿Cómo?

Divide y Vencerás

Número de módulos

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

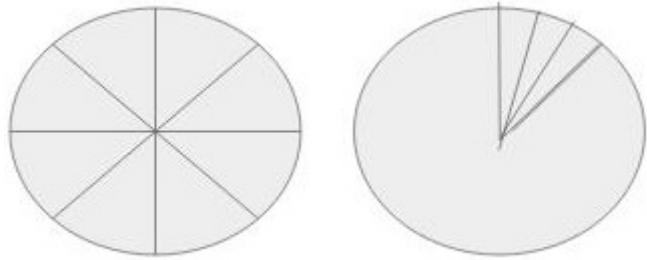
- **Costes de la modularización**, es un compromiso, un **equilibrio**, entre:
 - el **coste de desarrollo** de cada módulo, pocos muy grandes es más costoso que muchos muy pequeños
 - el **coste de integración** de todos los módulos, muy pocos cuesta poco y muchos cuesta mucho



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

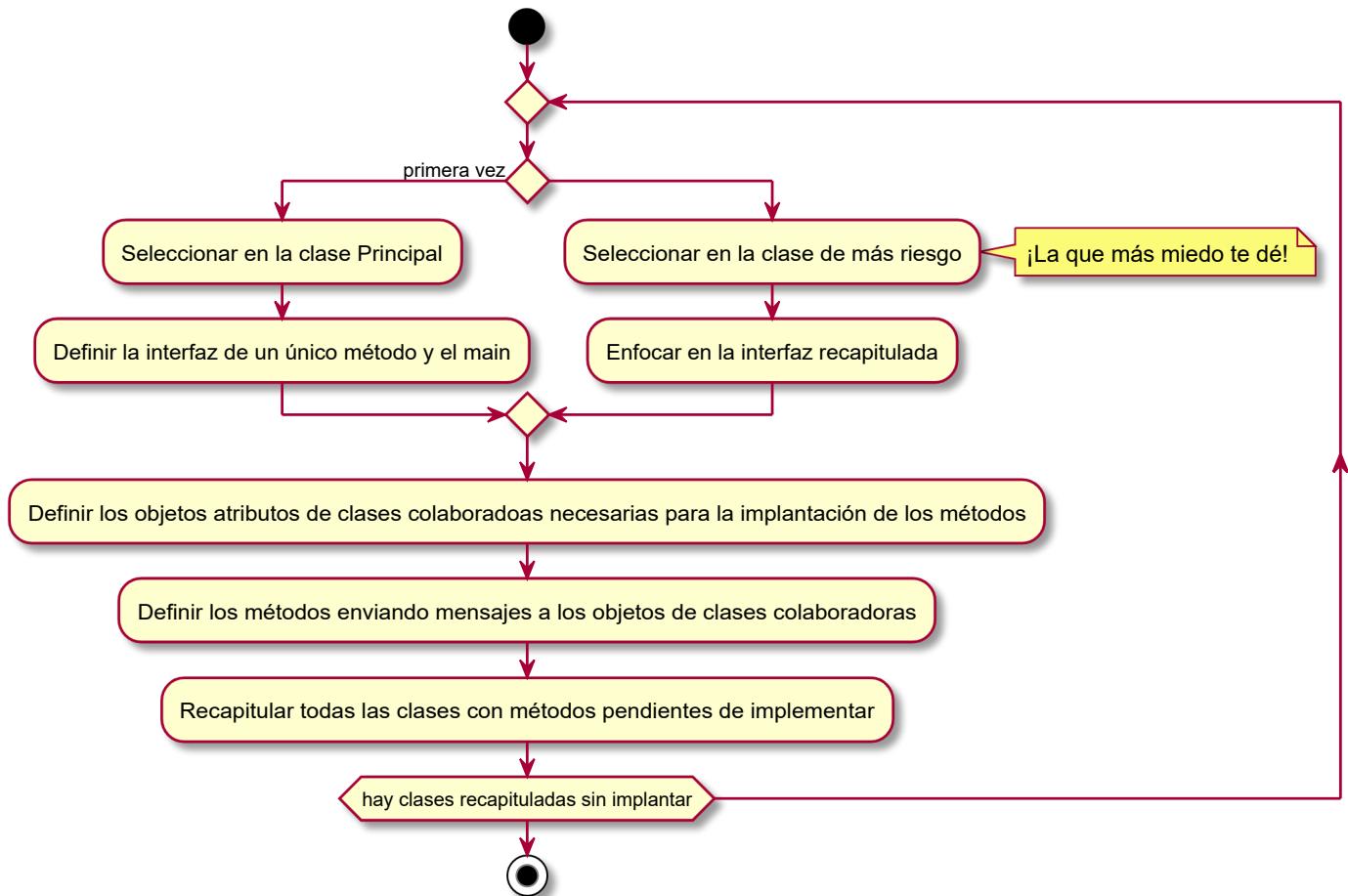
Distribución de Responsabilidades

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



Diseño dirigido por Niveles

- **Diseño descendente (top/down);**



- **Diseño ascendente (bottom/up):**

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

- **Comparativa**

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

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

Interfaz

Interfaz Suficiente, Completa y Primitiva

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

Principios del Menor Compromiso y la Menor Sorpresa

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

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

“Principio del menor compromiso, a través del cual la interfaz de un objeto proporciona su comportamiento esencial, y nada más

— Abelsony Sussman

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

— Booch

Código Sucio por Clases Alternativas con Interfaces Diferentes

- **Sinónimos:**

“Clases Alternativas con Diferentes interfaces (Alternative Classes with Different Interfaces)

— Smell Code (Refactoring); Martin Fowler

- **Justificación:** Complejidad innecesaria

- **Solución:**

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

Diseño por Contrato

- **La corrección** sólo tiene sentido en relación con una determinada especificación
 - Un **fallo** es cuando un sistema software **se aparta** de su comportamiento especificado durante una de sus ejecuciones
 - Un **defecto** es una propiedad de un sistema de software que pueden hacer que el sistema **se aparte** de su comportamiento especificado.
 - Un **error** es una **mala decisión** hecha durante el desarrollo de un sistema software que produce defectos
 - **Un fallo es el hecho real y un defecto es posibilidad potencial**
 - **Los fallos son debidos a los defectos los cuales resultan de los errores**
- **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 **fuerza de los límites del software** de la biblioteca.
- **Gestión de Errores:**
 - La robustez, la capacidad del software de reaccionar a casos no incluidos en la especificación, de los posibles errores excepcionales se cubre generalmente con excepciones.
 - *Por ejemplo: abrir un fichero no existente o sobre un soporte dañado, envío y recepción de datos sin conexión en red o con la base de datos, uso inadecuado de una biblioteca, ...*
 - La corrección, la capacidad del software de ejecutar de acuerdo con sus especificaciones, de los posibles errores lógicos se cubre generalmente **inadecuadamente con Programación defensiva y adecuadamente con Aserciones**
 - **Programación Defensiva:** para obtener software fiable se debe diseñar cada componente de un sistema de modo que se proteja a sí mismo tanto como sea posible.
 - La solución es que cada componente (método) compruebe la viabilidad de operar con *if-then-else*. Pero:
 - **No basta con informar por pantalla** del error lógico porque no se puede acoplar dicho componente a la vista con tecnologías alternativas (consola, gráfica, móvil, web, ...) y porque habrá que avisar al cliente para que tome las medidas oportunas ante el error
 - **No basta con un código de error** cuando no es posible acordar un valor particular de error (0 ó -1) si toda la gama es una posible solución
 - En caso optar por la Programación Defensiva **tanto el componente como su cliente aumentarán innecesariamente su complejidad** con sentencias *if-then-else* tanto para confirmar la viabilidad 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.

- **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 comprobaciones automáticas durante la ejecución y, en caso de error, elevar una excepción que termina la ejecución e informa claramente de lo que sucedió
- **Diseño por Contrato** es ver las relaciones entre una clase y sus clientes 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 comprensión del problema y sus eventuales soluciones
 - Facilitar la tarea de documentación del software
 - **Protocolo** es el conjunto entero de operaciones que un cliente puede realizar sobre un objeto junto con las “consideraciones legales” en los que pueden ser invocadas.
 - Para cada operación asociada con un objeto, se pueden definir precondiciones y postcondiciones: {P} A {Q}: donde A denota una operación; P y Q son aserciones sobre las propiedades de varias entidades involucradas; P es llamada precondición y Q postcondición.
 - Cualquier ejecución de A, comienza en un estado que cumple P y terminará en un 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 clientes no pueden confiar en el comportamiento del servidor
 - La pareja precondición/postcondición de una rutina describen el contrato que la rutina(servidor de un cierto servicio) define para sus usuarios (clientes del servicio)
 - **Las Precondiciones** atan al cliente con las restricciones sobre el estado de los parámetros y del objeto servidor que se deben cumplir para una llamada legítima a la operación y que funcione apropiadamente. Son una obligación para el cliente y un beneficio para el servidor.
 - **Precondiciones fuertes** exigen más al cliente para solicitar una tarea y facilitan el trabajo del servidor restringiendo las condiciones de partida
 - **Precondiciones débiles** exigen menos al cliente para solicitar una tarea pero **complican el trabajo del servidor** ante más amplitud en las condiciones de partida
 - **Las Postcondiciones** atan al servidor con las restricciones sobre el estado del valor devuelto y del objeto servidor que se deben cumplir tras el retorno de la operación para que el cliente progrese adecuadamente. Son una obligación para el servidor y un beneficio para el cliente:

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

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

- Una **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 diferente de las precondiciones y postcondiciones caracterizadas a rutinas individuales sobre sus parámetros de entrada y sus resultados respectivamente junto con el estado del objeto. La invariante solo involucra el estado del objeto.
 - Añadir Invariantes de Clase fortalece o mantiene como poco las precondiciones y postcondiciones porque la invariante:
 - **Facilita el trabajo del componente** porque además de la precondición, se puede asumir que el estado inicial del objeto cumple la invariante, lo que restringe el conjunto de casos que se deben contemplar
 - **Complica el trabajo del componente** porque además de la postcondición, se debe cumplir que el estado final del objeto cumpla la invariante, lo que puede aumentar las acciones a realizar
 - Una clase es **correcta** así:
 - **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: $\{P\} \text{ constructor } \{Q \text{ and } I\}$
 - **Cada operación** de la clase, cuando se aplica satisfaciendo su precondición y su invariante, cuando termina satisface su postcondición y su invariante: $\{P \text{ and } I\} \text{ operación } \{Q \text{ and } I\}$

Implementación

Cohesión

Cohesión de Métodos

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

- **Justificación:**

- A menudo se intenta crear funciones que tienen multiples secciones que realizan una serie de operaciones. Dicho de otra manera, la relación entre las líneas de la implantación del método no son cohesivas porque persiguen distintos objetivos
- Producen un acoplamiento temporal e imposibilitan su reusabilidad.
 - *Por ejemplo: Método que calcula la longitud de un Intervalo 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

Principio de Única Responsabilidad

- Definido por Robert Martin (*Single Responsibility Principle -SRP*) como uno de los principios SOLID
- Está inspirado en los trabajos de *De Marco y Page-Jones*, denominado como **cohesion**: relación funcional de los elementos de un modulo. Pero desplaza un poco el significado y relaciona la cohesion con la causa de 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
- **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 las tecnologías de la capa de persistencia en aquellas clases que manejan la entidad*

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

Código Sucio por Cambios Divergentes

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

- **Justificación:** Ocurre cuando una clase se cambia frecuentemente de diferentes maneras, por diferentes razones. Si nos fijamos en una clase y dice: "Bueno, voy a tener que cambiar estos tres métodos cada vez que tengo una nueva base de datos, tengo que cambiar estos cuatro métodos cada vez que hay un nuevo instrumento financiero, ..."
- **Solución:**

- Es probable que tenga una situación en la que varios objetos son mejor que uno. De esta manera cada objeto sólo se cambia como resultado de un tipo de cambio. Por supuesto, a menudo se descubre esto sólo después de añadir un par de bases de datos o instrumentos financieros.
- Estructuramos nuestro software para hacer el cambio más fácil. Despues de todo, el software está destinado a ser blando. Cuando hacemos un cambio queremos la ventaja de ser capaces de saltar a un solo punto en el sistema y hacer el cambio.

Código Sucio por Cirugía a Escopetazos

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

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

Código Sucio por Clase de Datos

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

- **Justificación:**

- Hay clases que tienen atributos, métodos *get/set* y nada más. Estas clases son soportes de datos tontos y es casi seguro que se manipulan con demasiado detalle por otras clases.
 - Las clases necesitan tomar alguna responsabilidad
- **Solución:**
- Buscar des de dónde se llaman los métodos *get/set* que son usados por otras clases. Intentar mover el comportamiento dentro de la clase de datos.
 - Después eliminar los métodos *_get/set_innecesarios*

Código Sucio por Envidia de Características

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

- **Justificación:** Un mal olor clásico es un método que parece más interesado en una clase distinta de la que realmente es. El enfoque más común de la envidia son los datos. Multitud de veces se ve un método que invoca media docena de métodos para conseguir calcular un valor de otro objeto.
- **Solución:**

- El método claramente quiere estar en otro lugar. A veces sólo una parte del método adolece de envidia; en ese caso, extraer el método ponerlo en la clase adecuada.
- La clave de los objetos es una técnica para empaquetar datos con los procesos utilizados en esos datos.
- Si se extrae información de objetos de varias clases combinadamente, colocar el método en la clase que más atributos aporta para el cálculo

Código Sucio por Clases Perezosas

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

- **Justificación:**
- Cada clase que se crea cuesta dinero para mantenerla y entenderla.
 - Una clase que no está haciendo lo suficiente para justificar el coste por sí mismo debería ser eliminada.
 - A menudo, esto podría ser una clase que paga por su bagaje y se ha reducido con la refactorización.
 - O podría ser una clase que fue añadida a causa de los cambios que estaban previstos, pero nunca llegaron.

- **Solución:**

- De cualquier manera, dejar que la clase muera con dignidad asignando su escasa responsabilidad a otra clase
- Si hay subclases que no están haciendo lo suficiente, trate de contraerla jerarquía.

Código Sucio por Obsesión por Tipos Primitivos

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

- **Justificación:** 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 assignable, puede crear un enumerado

Código Sucio por Grupo de Datos

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

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

Acoplamiento

Leyes de Demeter

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

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

Código Sucio por Librería Incompleta

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

- **Justificación:**

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

Inapropiada Intimidad

- **Justificación:** Una relación bi-direccional complica el desarrollo, las pruebas, la legibilidad, ...
- **Solución:**
 - 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 poniéndolo común a salvo y haz que las demás sean honestas sobre ella.
 - La herencia a menudo puede conducir 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 privadas de otras clases. Nosotros pensamos que nuestras clases deberían ser estrictas con reglas puritanas

— Fowler
Refactoring. 1999

Tamaño

Código Sucio por Listas de Parámetros Largas

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

- **Justificación:**
 - Son difíciles de entender
 - Son difíciles de probar todas las 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 asigne responsabilidad a sus clase
- **Métrica:**
 - Funciones deberán tener un número pequeño de argumentos. Sin argumentos es lo mejor, seguido por uno, dos. Tres debería evitarse y más de tres es muy cuestionable y debe considerarse como un prejuicio.

Código Sucio por Métodos Largos

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

- **Justificación:**

- Desde los principios de la programación, los programadores se han dado cuenta de que cuanto más largo es un procedimiento, más difícil es de entender. Los viejos lenguajes llevaban una sobrecarga en las llamadas a subrutinas, de tal forma que se persuadía de escribir métodos pequeños.

- **Solución:**

- El 99% de las veces, se tiene que acortar un método extrayendo otro. Buscar una parte del método que parezca ir bien junta y hacerlo un nuevo método
- Una buena técnica es mirarlos comentarios o líneas en blanco para separar partes. Son señales de esta 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

- **Métrica:**

- Número de Lineas: [10, 15] como máximo
- Caracteres por línea: [80,120] como máximo
- Complejidad ciclomática: [10-15] como máximo

- **Implicaciones:**

- Los nuevos programadores orientados a objetos a menudo sienten que la computación no se hace en ninguna parte, que los programas son secuencias sin fin de delegación. Cuando has vivido con un programa como tal por unos años, aprendes cómo de valorable son todos esos pequeños métodos. Todos los costes de indirección-explicación, compartición y selección –son respaldadas por pequeños métodos

Código Sucio por Clases Grandes

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

- **Justificación:**

- Cuando una clase está tratando de hacer demasiado, a menudo aparece con demasiadas variables de instancia. En tal caso, el código duplicado no puede estar muy lejos.
- Los archivos pequeños son generalmente más fáciles de entender que archivos de gran tamaño.

- **Métrica:**

- Parece ser posible construir sistemas significativos con archivos que son típicamente de 200 líneas de largo, con un límite máximo de 500. A pesar de que esto no debería ser una regla dura y rápida, debe considerarse muy deseable.
- 3 atributos de media; 5 como máximo
- 20 métodos como máximo

- **Solución:**

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

Código Sucio por Atributos Temporales

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

- **Justificación:**

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

- **Solución:** En este caso, se puede extraer en una clase los atributos y los métodos que lo requieran. El nuevo objeto es un *objeto método* [Beck]

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

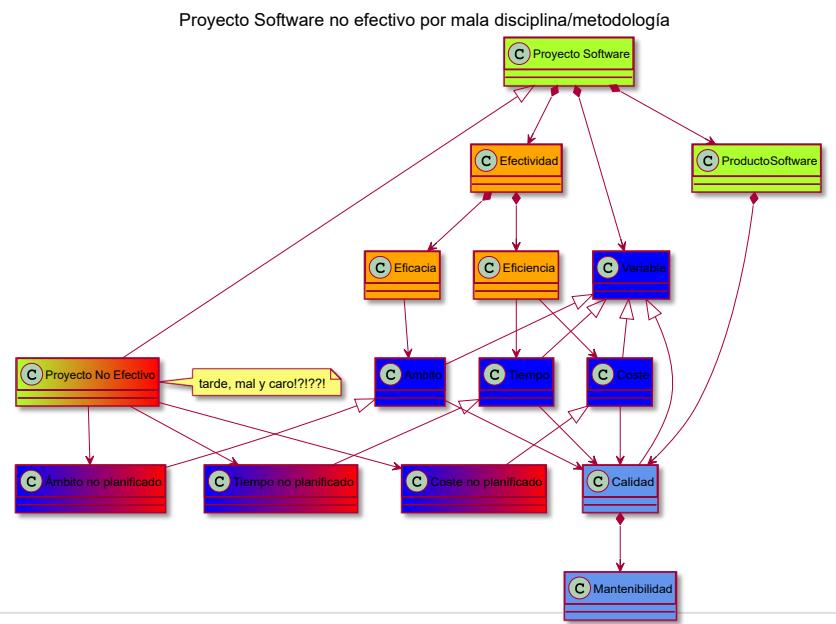
Principio de Sustitución de Liskov

Herencia vs Delegación

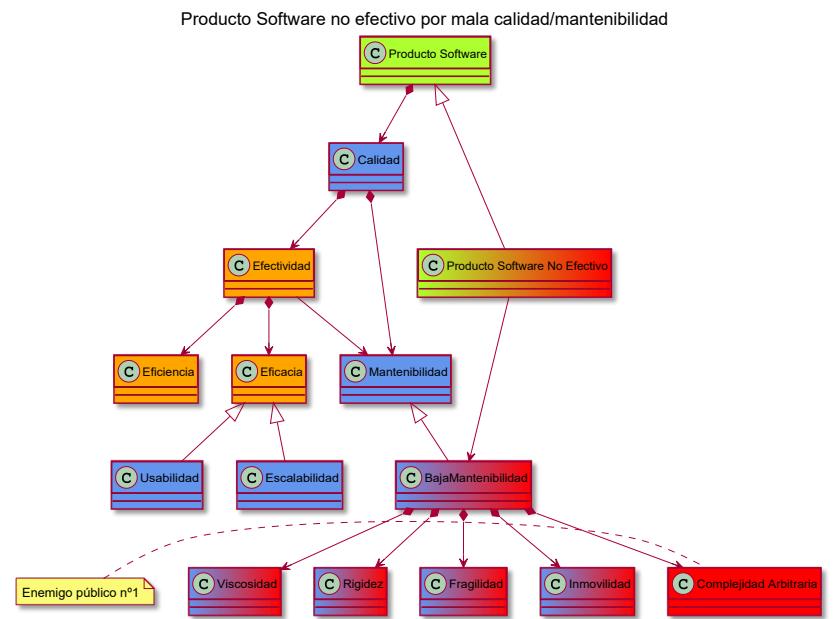
Bibliografía

Justificación: ¿Por qué?

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

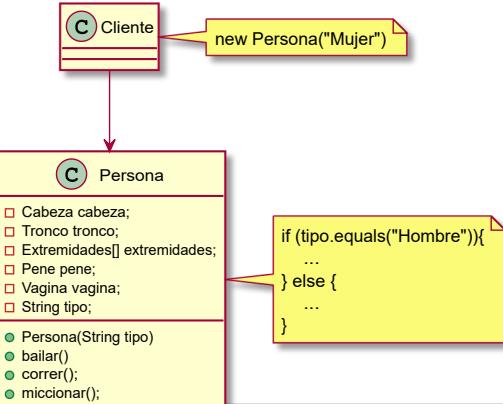
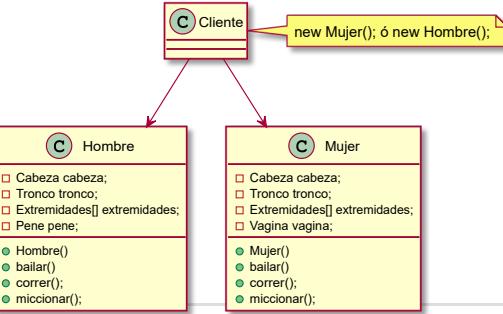
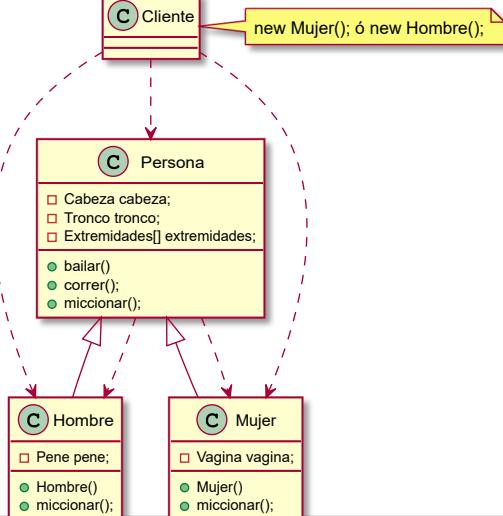


- Producto Software poco efectivo
 - porque tiene **mala calidad**
 - **Poco eficiente**
 - Poco eficaz en **corrección, usabilidad, escalabilidad, ...**
 - porque tiene **mala mantenibilidad**, de la que depende la eficiencia y la eficacia anteriores, porque es
 - **viscoso**, porque no se puede entender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



- El resultado del diseño modular **no asegura un código mantenable, 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
Una única clase asume la responsabilidad de toda la jerarquía	 <pre> classDiagram class Cliente { new Persona("Mujer") } class Persona { Cabeza cabeza; Tronco tronco; Extremidades[] extremidades; String tipo; Persona(String tipo); bailar(); correr(); miccionar(); } if (tipo.equals("Hombre")){ ... } else { ... } </pre>	<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"
Existe una clase por cada tipo de elemento que asumen sus correspondientes responsabilidades	 <pre> classDiagram class Cliente { new Mujer(); ó new Hombre(); } class Persona { Cabeza cabeza; Tronco tronco; Extremidades[] extremidades; String tipo; Persona(String tipo); bailar(); correr(); miccionar(); } class Hombre { Pene pene; Hombre(); bailar(); correr(); miccionar(); } class Mujer { Vagina vagina; Mujer(); bailar(); correr(); miccionar(); } </pre>	<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, ...
Existe una jerarquía de clases por cada tipo de elemento que asumen sus correspondientes responsabilidades	 <pre> classDiagram class Cliente { new Mujer(); ó new Hombre(); } class Persona { Cabeza cabeza; Tronco tronco; Extremidades[] extremidades; bailar(); correr(); miccionar(); } class Hombre { Persona; Pene pene; Hombre(); miccionar(); } class Mujer { Persona; Vagina vagina; Mujer(); miccionar(); } </pre>	<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
<p>enlace que se puede resolver en tiempo de compilación, característica que se puede determinar mirando el código</p> <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 ... 	<p>enlace que solo se puede resolver en tiempo de ejecución, característica que se puede determinar en un instante de la ejecución del código</p> <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 FFFFFFFFFFFFFFh? // 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
<p>Enlace estático, para toda expresión puede deducirse el tipo del valor del resultado de su evaluación en tiempo de compilación</p>	<p>Enlace dinámico, existen expresiones para las que no puede deducirse el tipo del valor del resultado de su evaluación en tiempo de compilación</p>
<p>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</p>	<p>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</p>

Este diagrama ilustra el enlace estático. Muestra una jerarquía de clases: X es la superclase, que contiene métodos m() y m(A, C). A y B son subclasses de X, cada una con su propia implementación de m(). C es otra clase que no hereda de X. Una variable X oX apunta a un objeto de tipo X. La ejecución de oX.m() se resuelve en el método de X, ya que el tipo de la referencia es conocido en tiempo de compilación.

```

    graph TD
        X["Clases X  
void m()  
A m(A)  
void m(A, C)  
A m(C, B)  
B m(B)"]
        A["Clase A"]
        B["Clase B"]
        C["Clase C"]
        oX["oX = new X();"]
        mX["oX.m(); // void"]
        mOA["oX.m(oA); // A"]
        mOC["oX.m(oA, oC); // void"]
        mOB["oX.m(oC, oB); // A"]
        mOB2["oX.m(oB); // B"]

        X --> A
        X --> B
        X --> C
        oX --> mX
        mX --> mOA
        mOA --> oA["oA = new A();"]
        mOA --> oB["oB = new B();"]
        mOC --> oC["oC = new C();"]
        mOB --> oX["oX = new X();"]
        mOB2 --> oB2["oB = new B();"]
    
```

Este diagrama ilustra el enlace dinámico. La jerarquía de clases es similar, pero la variable X oX ahora apunta a un objeto de tipo A (por ejemplo, oA). Al ejecutar oX.m(), el sistema evalúa el tipo real del objeto (A) en tiempo de ejecución y llama al método m(A) de la clase A, independientemente del tipo declarado en la referencia.

```

    graph TD
        X["Clases X  
void m()  
A m(A)  
void m(A, C)  
A m(C, B)  
B m(B)"]
        A["Clase A"]
        B["Clase B"]
        C["Clase C"]
        oX["oX = new X();"]
        mX["oX.m(); // void"]
        mOA["oX.m(oA); // A"]
        mOC["oX.m(oA, oC); // void"]
        mOB["oX.m(oC, oB); // A"]
        mOB2["oX.m(oB); // B"]

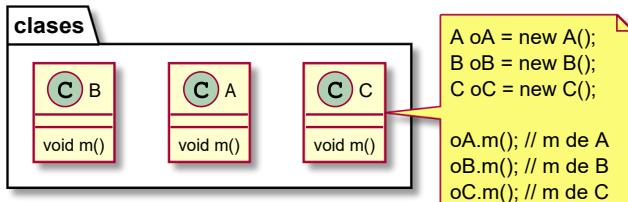
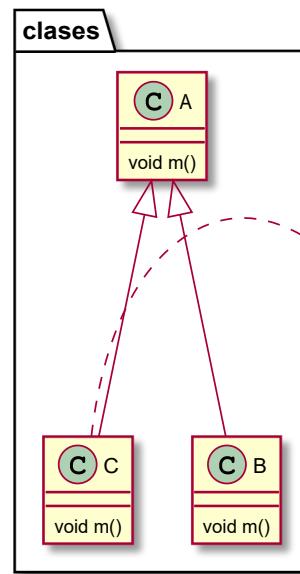
        X --> A
        X --> B
        X --> C
        oX --> mX
        mX --> mOA
        mOA --> oA["oA = new A();"]
        mOA --> oB["oB = new B();"]
        mOC --> oC["oC = new C();"]
        mOB --> oX["oX = new X();"]
        mOB2 --> oB2["oB = new B();"]
    
```

- 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)?

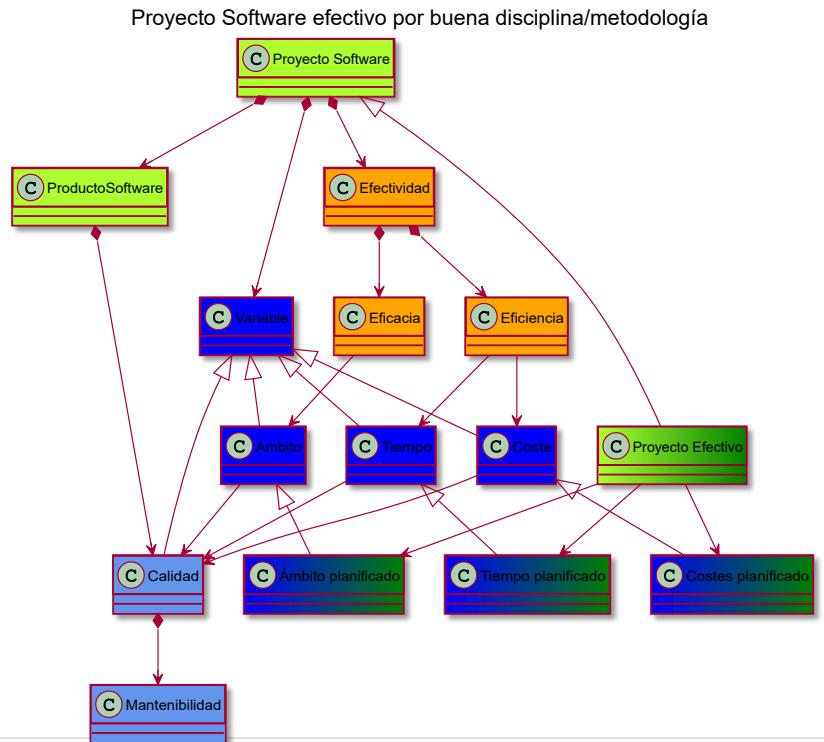
Sin polimorfismo	Con polimorfismo
<p>Enlace estático, un mensaje lanzado a un objeto ejecutará el método con el mismo nombre y parámetros, en número y tipos correspondientes, de la clase del objeto</p>	<p>Enlace dinámico, un mensaje lanzado a un objeto polimórfico ejecutará un método con el mismo nombre y parámetros, en número y tipos correspondientes, de alguna de las clases de la jerarquía a partir de la clase base de la referencia</p>
<p>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</p>	<p>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</p>

Sin polimorfismo**Con polimorfismo**

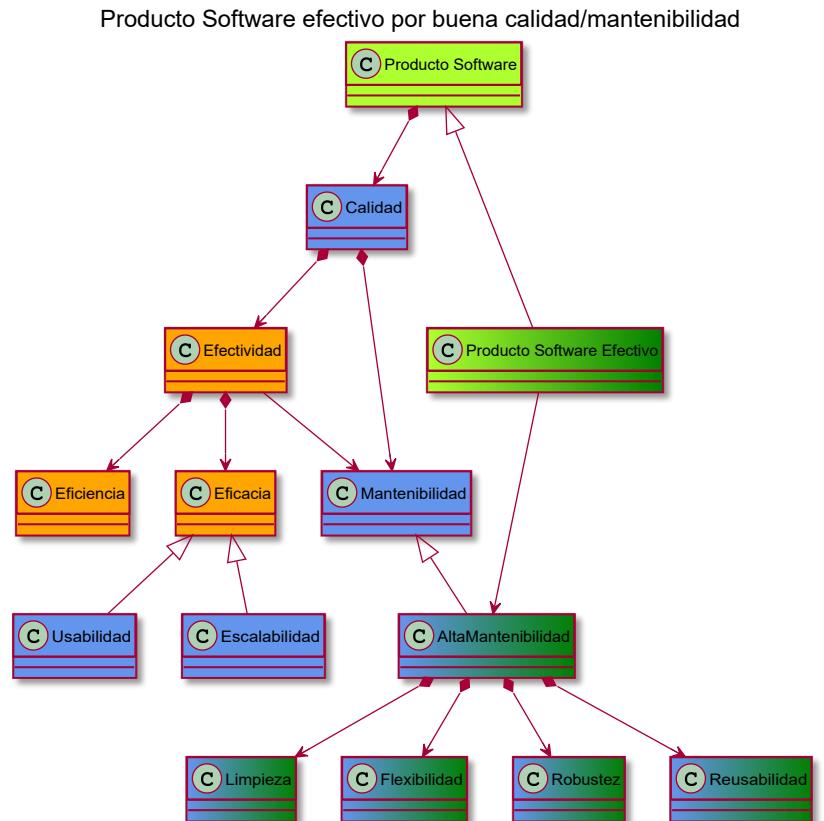
- Por tanto, se define el **polimorfismo como enlace dinámico de mensajes a métodos ejecutados**

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 **buenas variables**
 - **calidad**
 - Es eficiente
 - Es eficaz en **corrección, usabilidad, escalabilidad, ...**
 - porque tiene **buenas variables**, 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!**

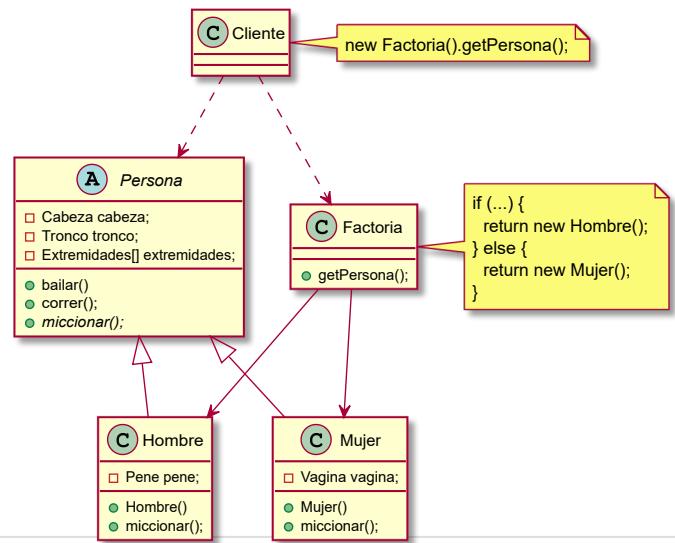
Flexible	Reparto de responsabilidades equilibrado y centralizado en clases que requiere modificarse únicamente si cambian los requisitos correspondientes, jerarquías de clases con alta cohesión, sin ciclos!
Resuable	Presencia de multitud de clases pequeñas, cohesivas y poco acopladas a tecnologías, algoritmos, ...!
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/Closed 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 extensión 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 todos 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 prescincia 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>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>
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 (@Override)
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 hables con extraños	Do not talk to strangers		Lieberherr
Cadena de Mensajes	Chain of Message	Smell Code (Refactoring)	Martin Fowler

- **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> ▪ 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 instancia 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() { //aaaaaaaaaaaa //yyyyyyyyyyyy //bbbbbbbbbbbb } } class Z extends X { public void m() { //aaaaaaaaaaaa //zzzzzzzzzzzz //bbbbbbbbbbbb } }</pre>	<p style="text-align: center;">JAVA</p> <pre>class X { public void m() { //aaaaaaaaaaaa this.middle(); //bbbbbbbbbbbb } } public abstract middle(); class Y extends X { public middle(){ //yyyyyyyyyyyy } } class Z extends X { public middle(){ //zzzzzzzzzzzz } }</pre> <p style="text-align: center;">JAVA</p>

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, pudiéndose 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 escribe en PrintWriter (módulos de bajo nivel), debería leer de una interfaz Reader, base de KeyboardReader y escribir en una interfazWriter, 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 data-bbox="112 1094 726 1749"> class KeyboardReader{ public char[] read(){ ... } } class PrintWriter{ public void write(char[]){ ... } } class Copier { public Copier(KeyboardReader reader, PrintWriter writer){ ... char[] data = reader.read(); ... writer.write(data); ... } } </pre> <p style="text-align: right;">JAVA</p>	<pre data-bbox="826 1094 1382 1964"> interface Reader { char[] read(); } class KeyboardReader implements Reader{ public char[] read(){ ... } } interface Writer { void write(char[]); } class PrintWriter implements Writer{ public void write(char[]){ ... } } class Copier { public Copier(Reader reader, Writer writer){ ... char[] data = reader.read(); ... writer.write(data); ... } } </pre> <p style="text-align: right;">JAVA</p>

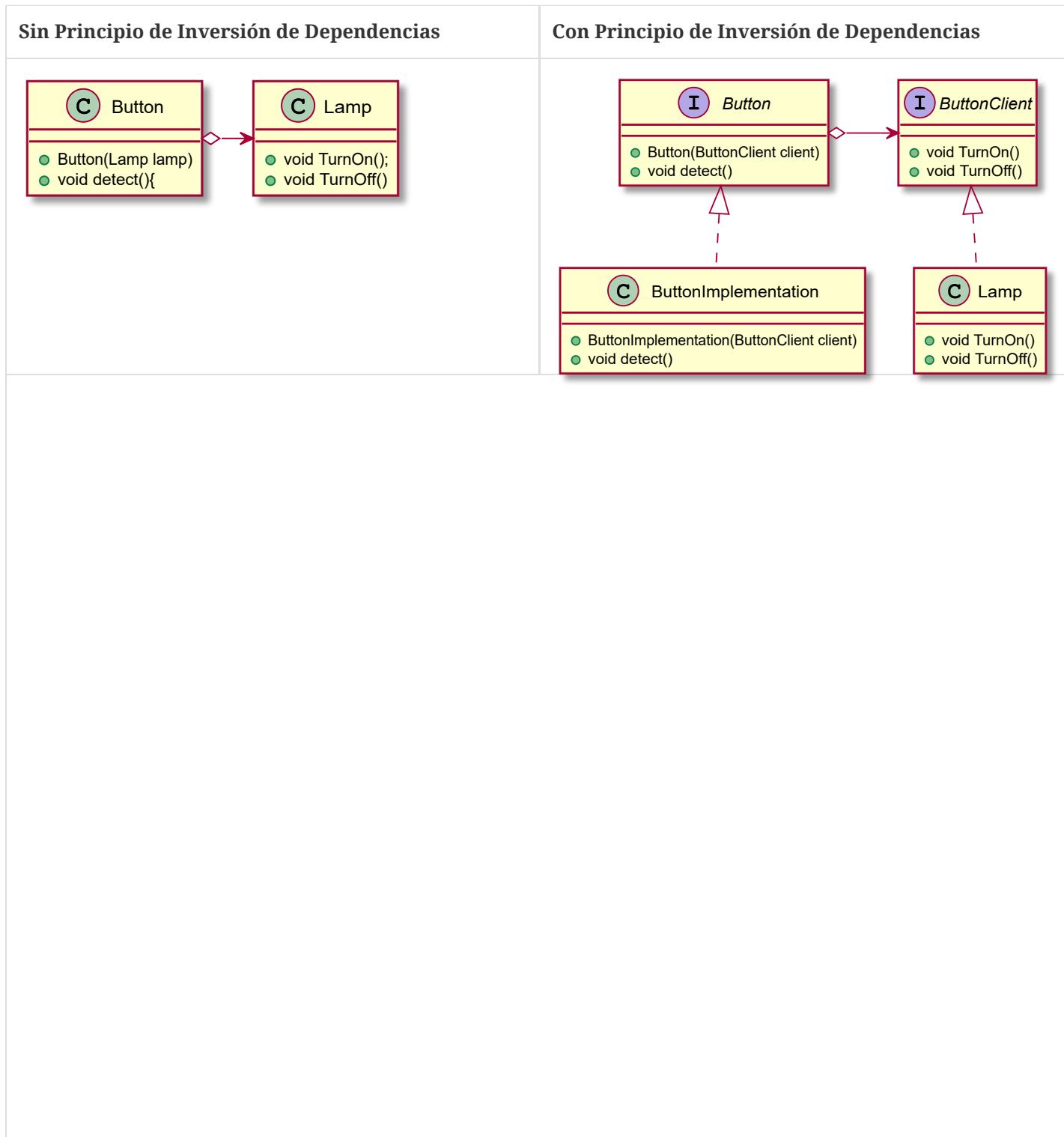
- Cuando los módulos de alto nivel son independientes de los de bajo nivel, se pueden reutilizar los primeros con sencillez. En este caso, **la instanciación de las objetos de bajo nivel dentro de la clase de alto nivel no puede ser hecha con el operador new**

- Por ejemplo: la lógica de Copier será reutilizada sin cambios cuando nuevos dispositivos de entrada y salida entran 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 data-bbox="73 159 787 1215"> 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 data-bbox="787 159 1504 1215"> 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 applicable a ciegas en cada clase o cada módulo**. Si se tiene la funcionalidad de una clase que es más que posible 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 calificación){ ... } 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> JAVA interface SecretariaAlumnos { void setCalificación(int id, int calificación); } interface SecretariaProfesores { int getIngresosFamiliares(int id); } class Secretaria implements SecretariaAlumnos, SecretariaProfesores { public void setCalificación(int id, int calificación){ ... } 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 una 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 ”

— Johnson & 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 ”

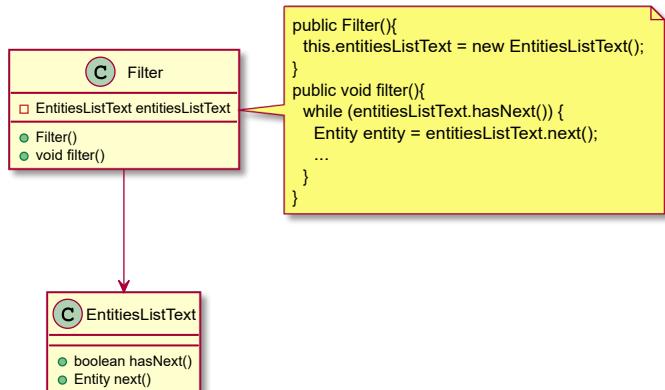
- **Variaciones:**

- **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

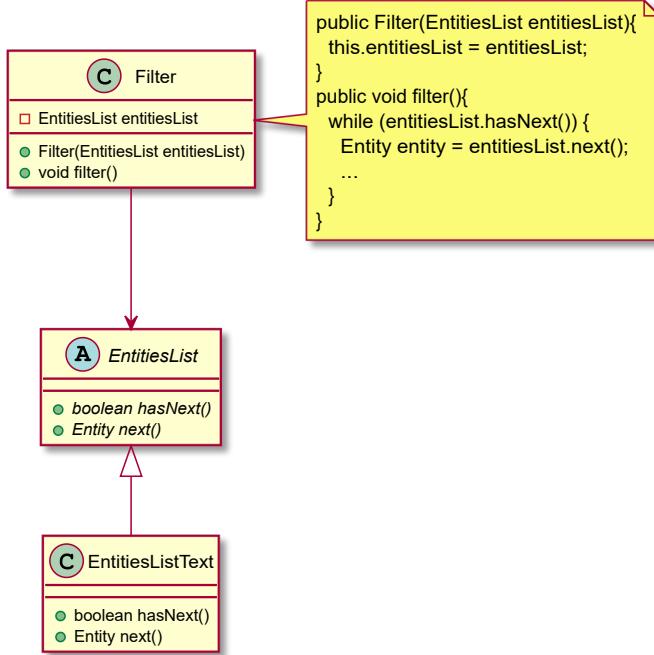


- **Solución:**

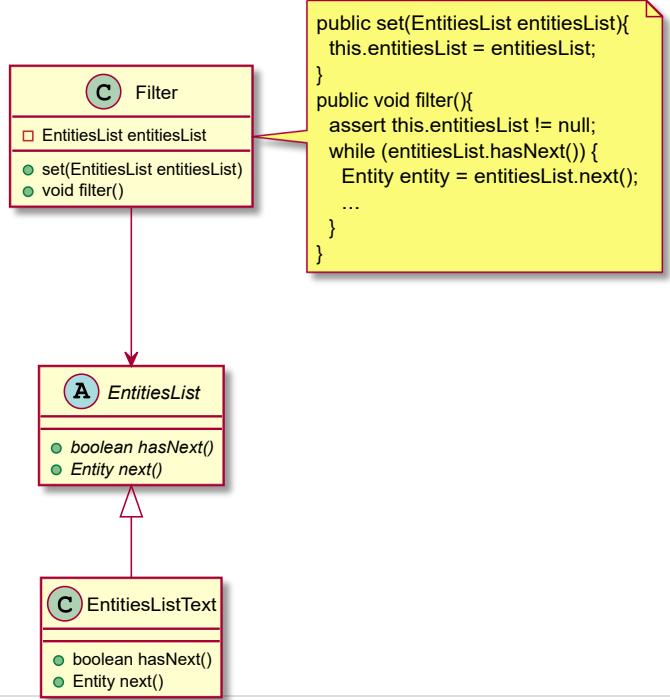
- Lo primero será que la **clase trabaje con un interfaz del servicio para que éste pueda ser extendido por otras implementaciones de servicio diferentes**. Pero, aunque la clase guarde la referencia al objeto servicio a través de un interfaz, mientras la clase instancie directamente el objeto servicio, 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.

Con inyección de dependencias por constructor



Con inyección de dependencias por método



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>Pinguino 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

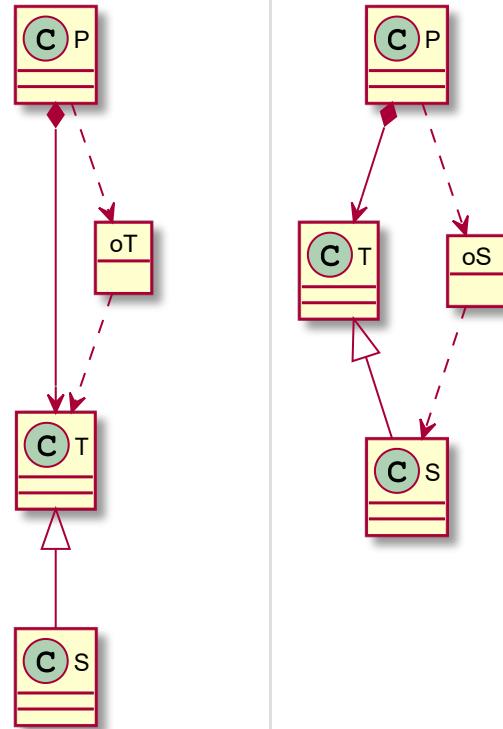
Principio de Sustitución de Liskov

- 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 **usen 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 precondición por una más débil y su postcondición por una más fuerte*”

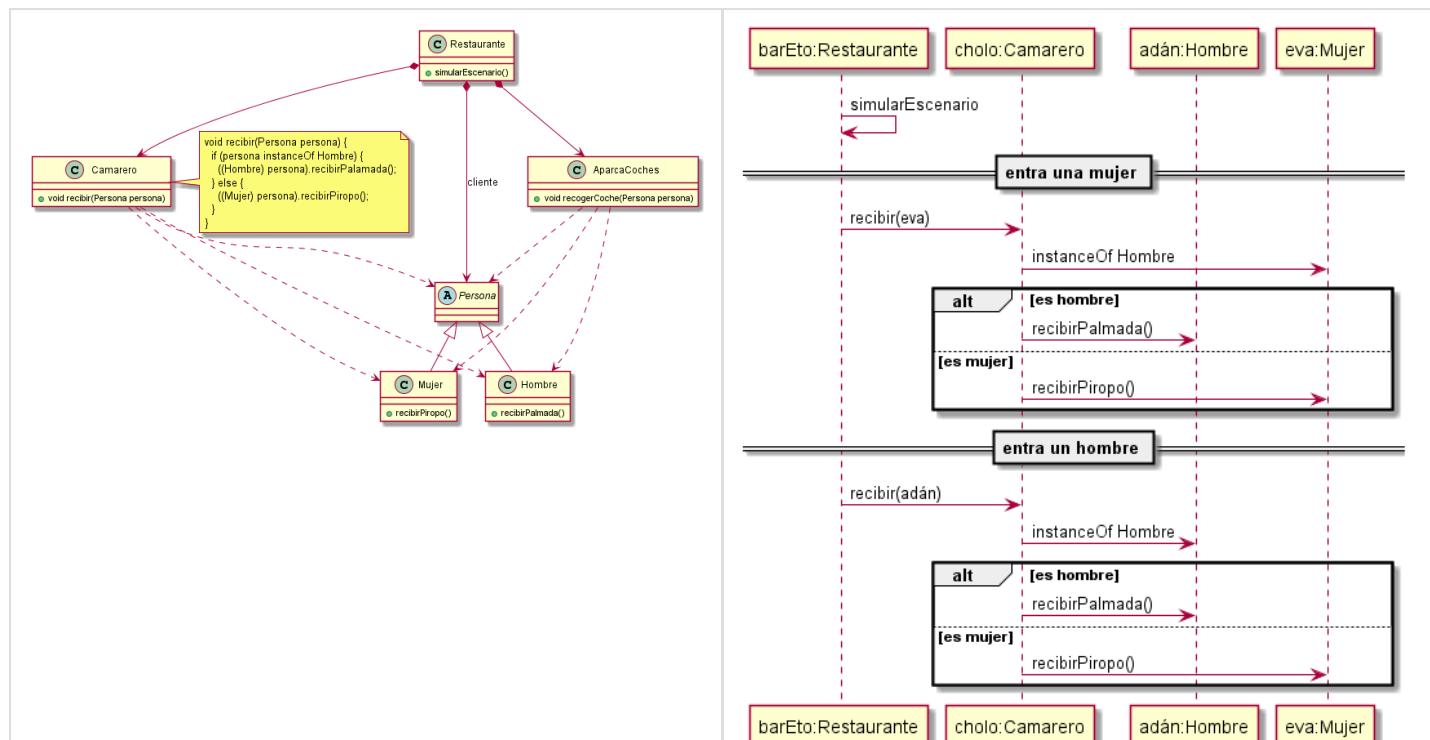
— Barbara Liskov
Principio de Sustitución

- Condiciones:
 - 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**.
- 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; ...*
- 1^a 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 1^a solución:**

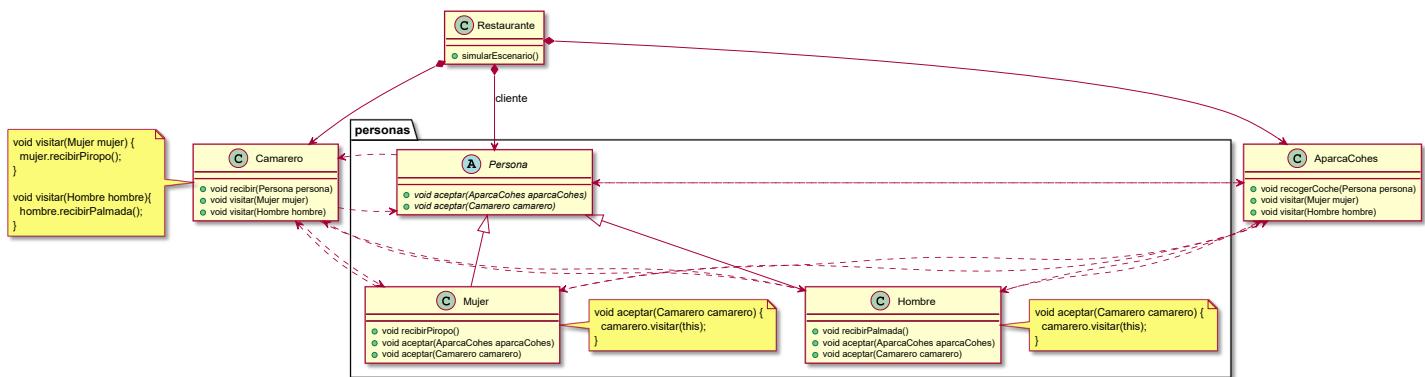
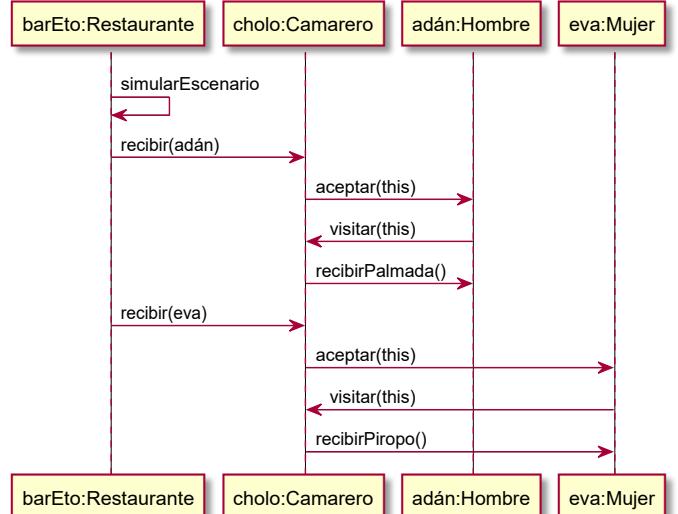
- viola el Principio de Sustitución de Liskov** preguntando por el tipo de objeto polimórfico
- incurre en cambios divergentes** para atender con una nueva rama en cada clase cliente que hay que localizar por toda la aplicación

- rompe el principio Open/Close con cambios en el interior de los métodos del cliente

- 2^a solución: aplicando la técnica de doble despacho (ver código)

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

- El cliente envia un mensaje **aceptar** al objeto polimórfico auto-pasándose como parámetro (*this*)
- Cada clase derivada devuelve un mensaje **visitar** al propio cliente auto-pasándose como parámetro (*this*)
- El cliente atiende por separado con métodos **visitar** para cada tipo de clase derivada con el comportamiento particular para cada uno



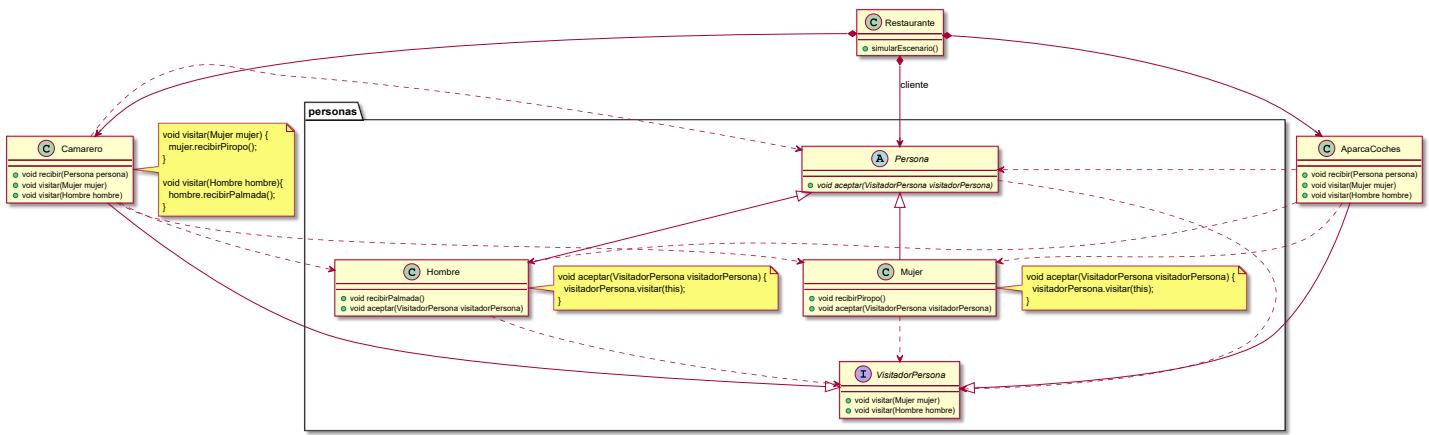
- Consecuencias de la 2^a solución:

- 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-pasándose 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
- intimididad 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

- 3^a solución: Principio de Inversión de Dependencias (ver código)

(<https://github.com/miw-upm/TWVG/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



- Consecuencias de la 3^a solución:

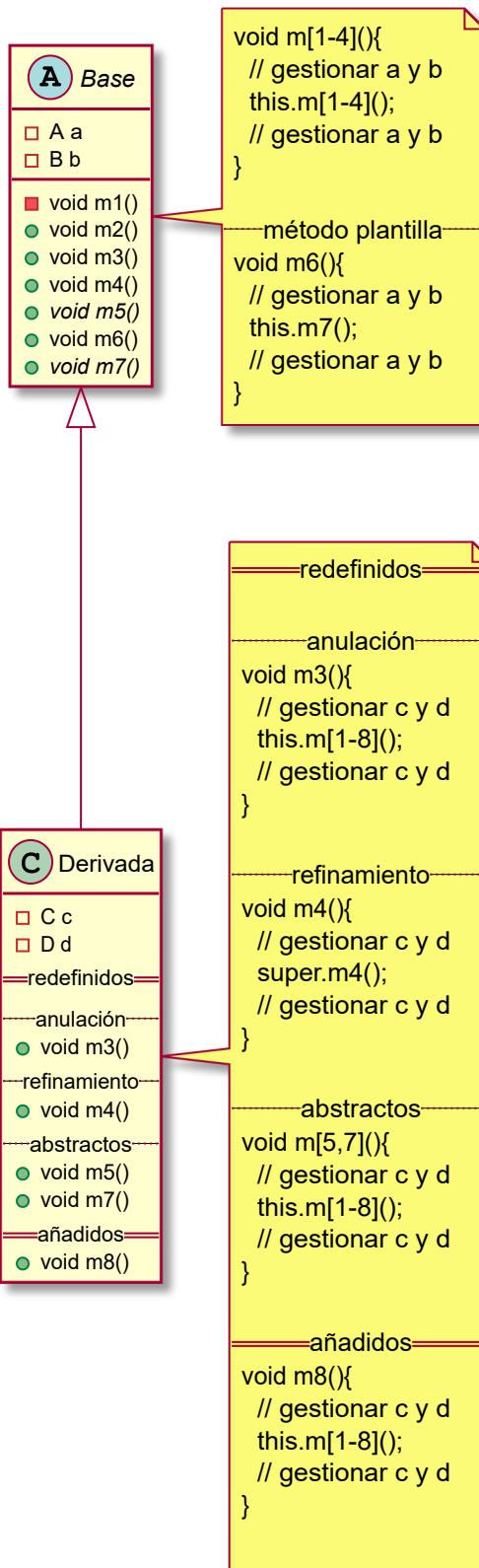
- no viola el Principio de Sustitución de Liskov preguntando por el tipo de objeto polimórfico
- no incurre en cambios divergentes para atender con una nueva rama en cada clase cliente
 - la nueva clase derivada debe redefinir el método **aceptar** para no ser abstracta enviando un mensaje *visitar* auto-pasándose 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
- con "leve" intimidad inapropiada con ciclos dentro del mismo paquete entre todas las clases de la jerarquía con la interfaz de los clientes, que no requiere pruebas ni comprensión porque no aporta código de implementación
- bajo acoplamiento según tienden a crecer los clientes porque no todas las clases de la jerarquía conocen a todas las clases de clientes, solo conocen a la interfaz de todos los clientes

Herencia vs Delegación

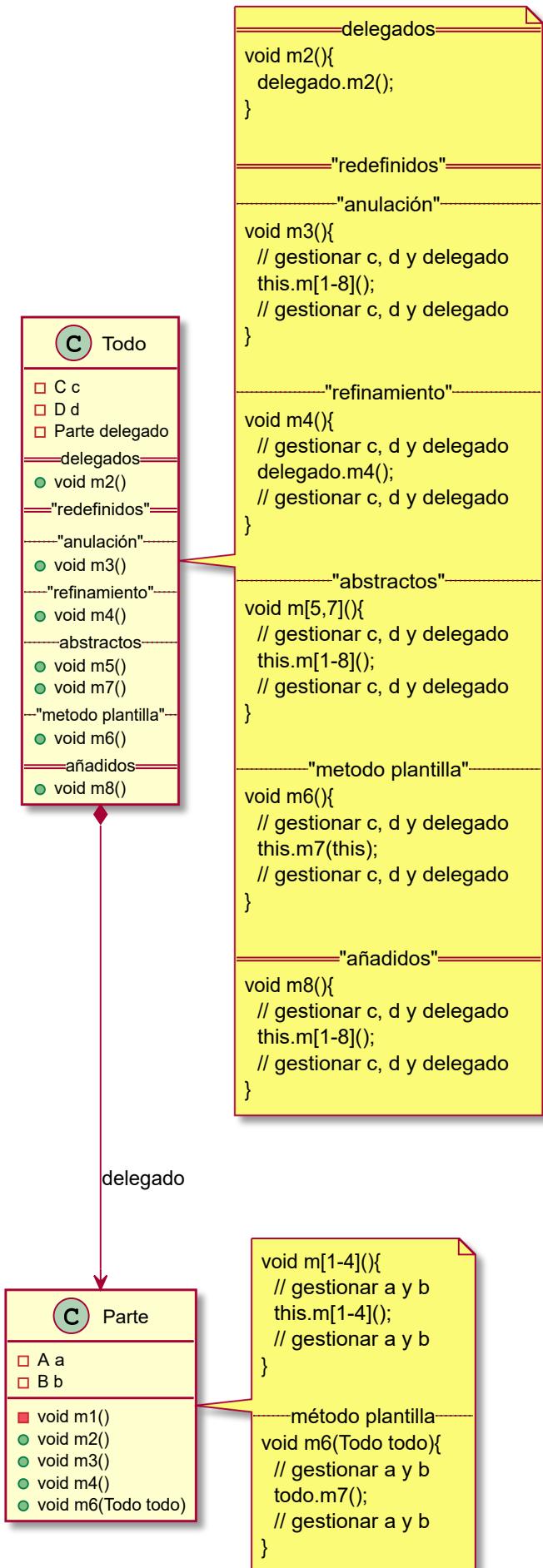
- Cualquier relación de herencia puede convertirse en una relación de composición

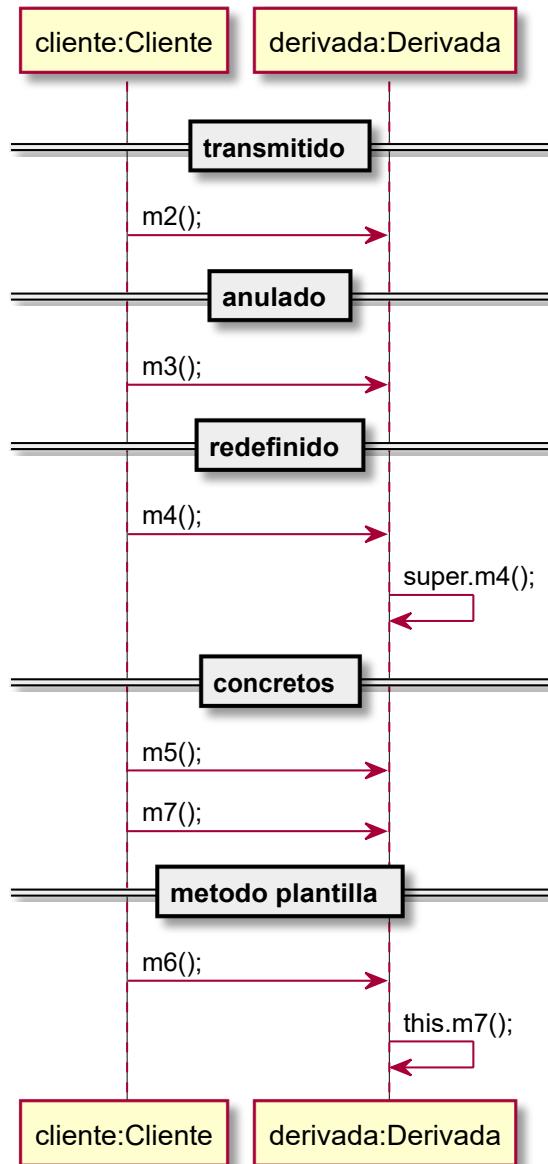
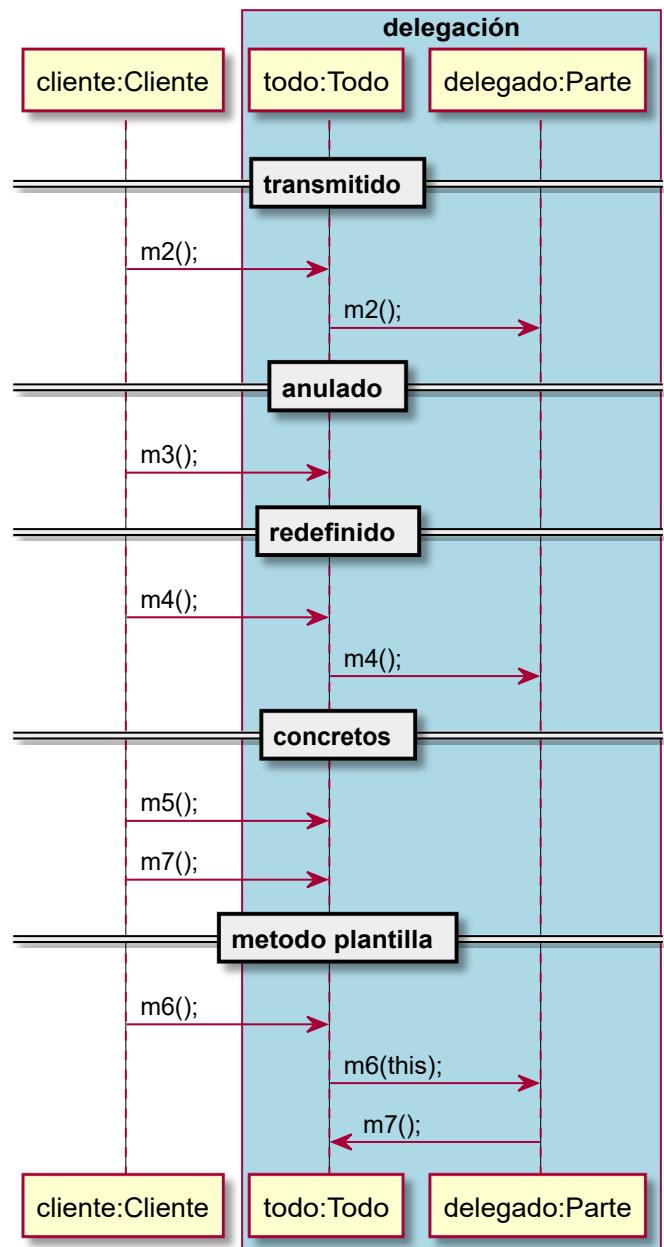
Relación de Herencia	Relación de Composición para Delegación

Relación de Herencia



Relación de Composición para Delegación

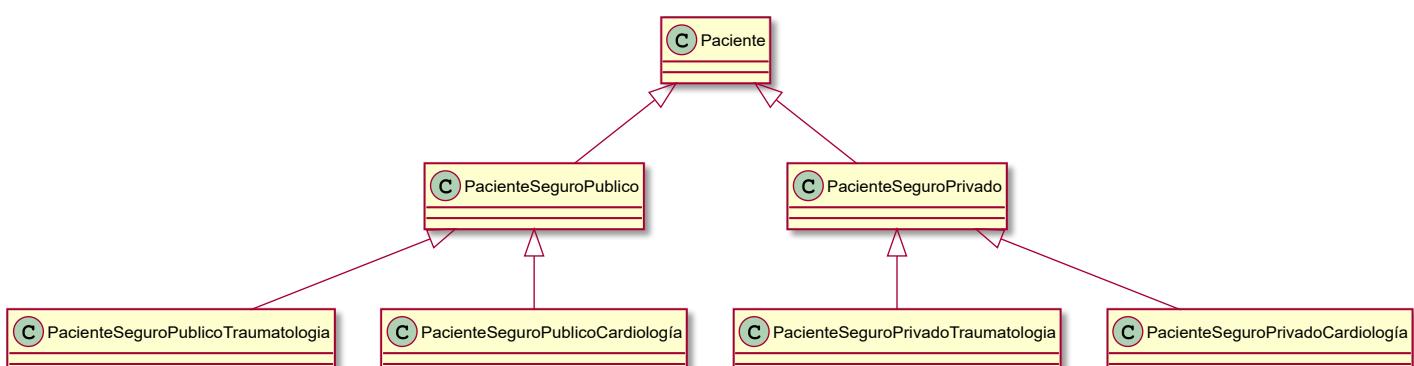


Relación de Herencia**Relación de Composición para Delegación**

Código Sucio por Jerarquías Paralelas de Herencia

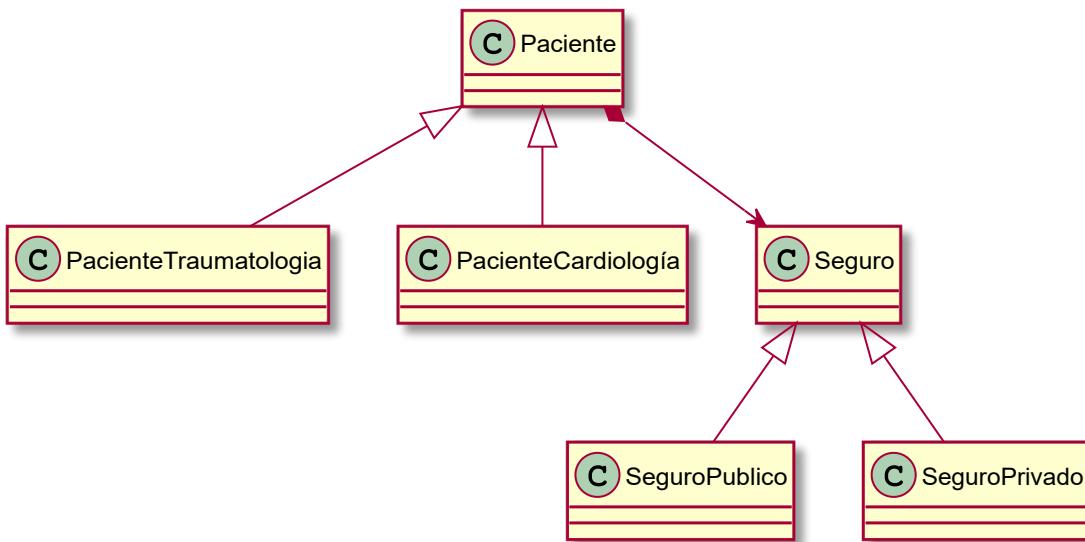
- Justificación:**

- 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

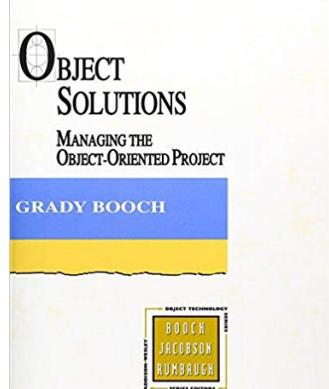
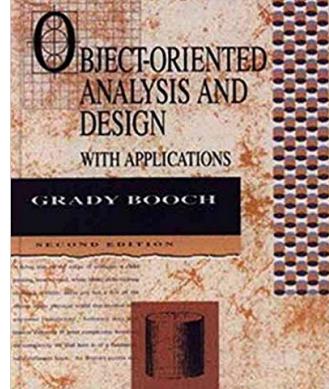
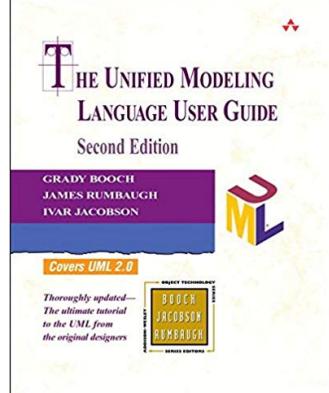
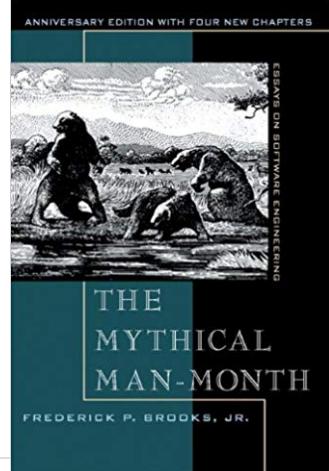
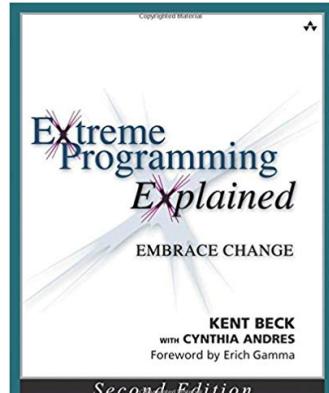
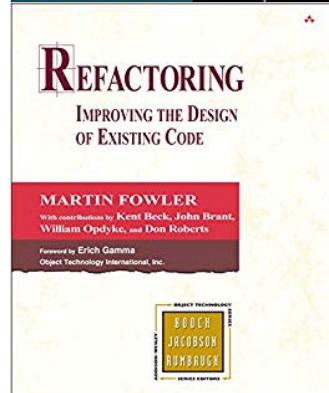


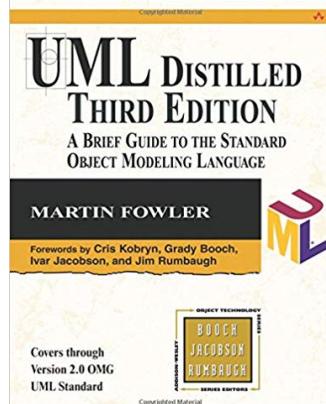
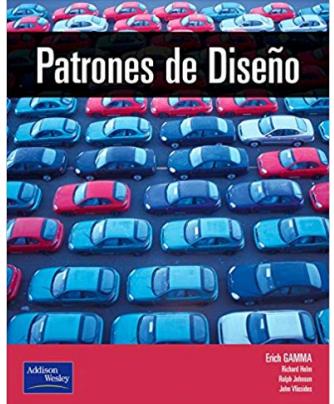
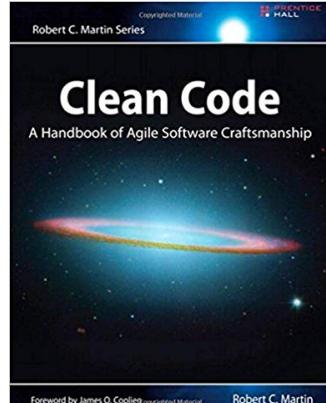
- **Solución:**

- Reestructurar la jerarquía:
 - Reubicar responsabilidades
 - Aplicar el Patrón Método Plantilla
 - Una clase contiene tantos roles polimórficos como cada una de las jerarquías 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 <ul style="list-style-type: none"> 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 ◦ 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 ◦ 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