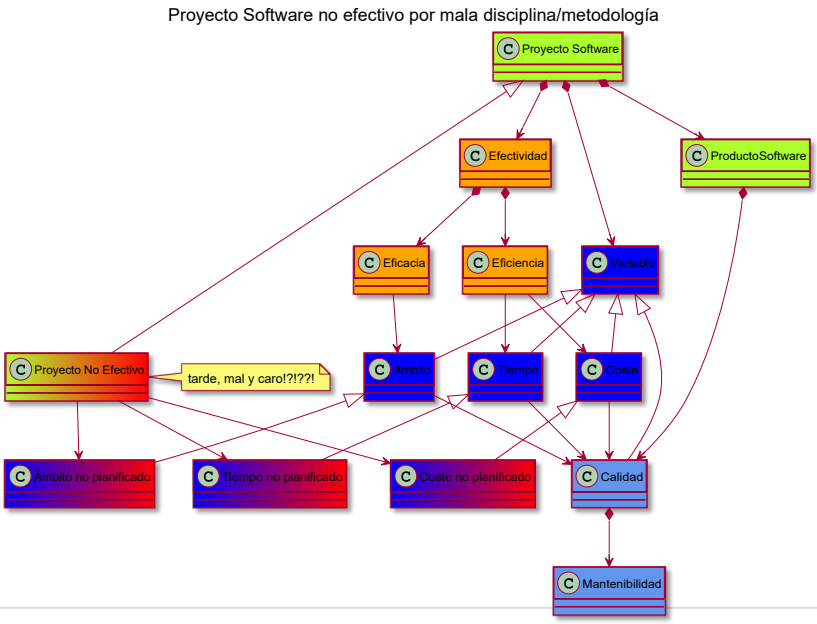
Diseño

Santa Tecla
parqueNaturalSantaTecla@gmail.com
Version 0.0.1

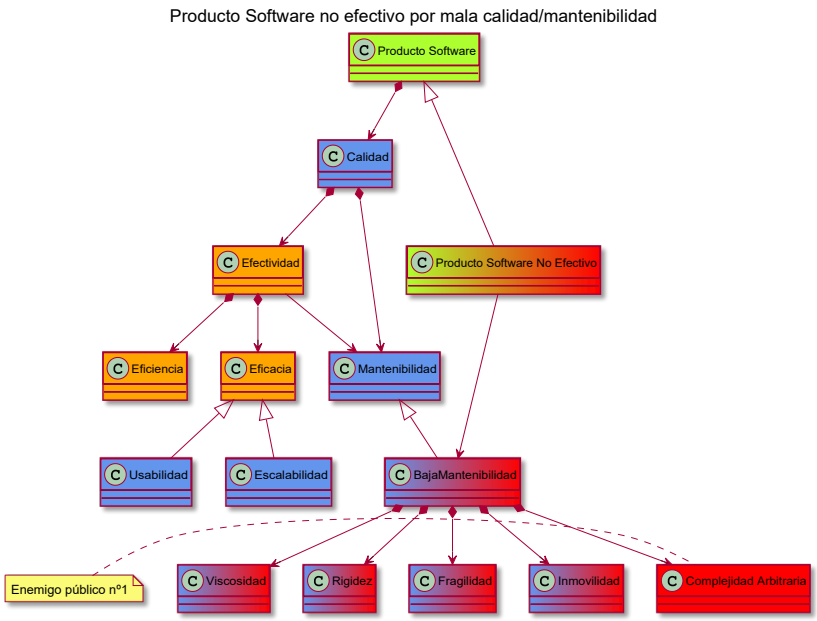
Índice

Justificación: ¿Por qué?

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



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



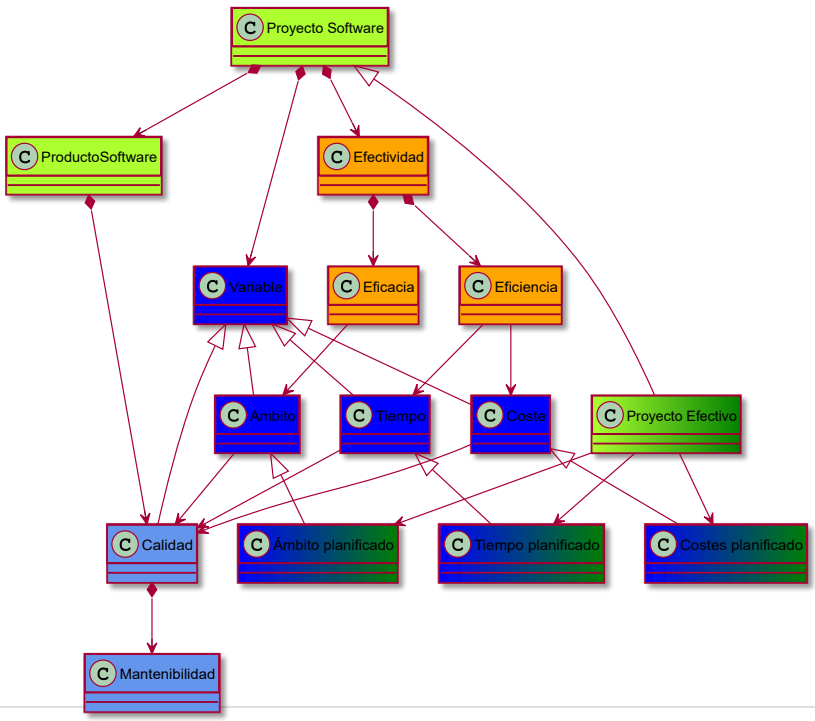
Definición: ¿Qué?

- 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?

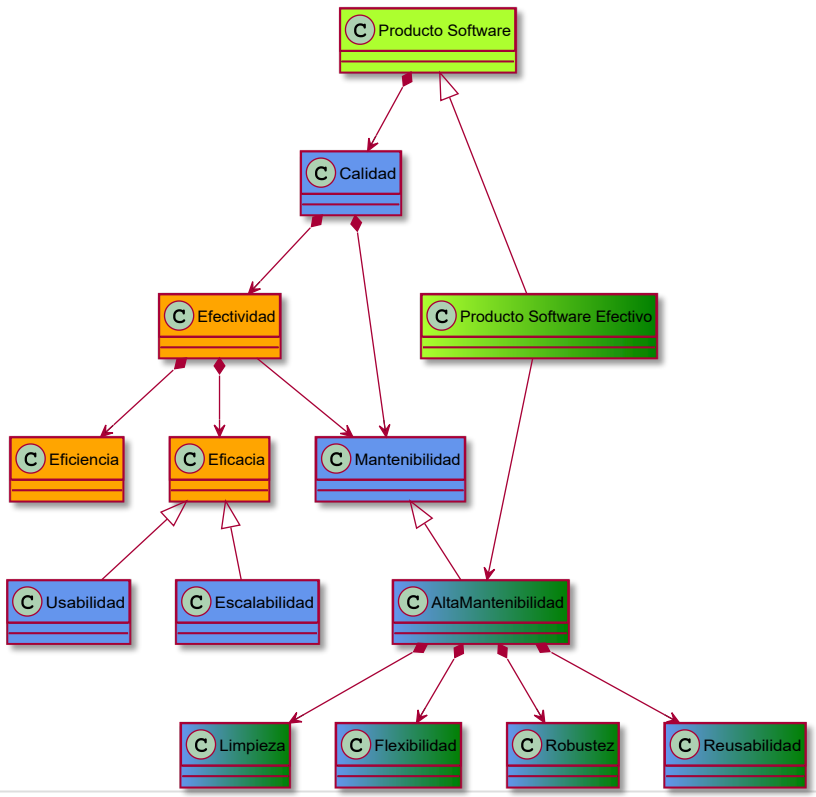
- Proyecto Software efectivo
  - porque tiene **buenas variables**
    - **tiempo cumplido**,
    - **ámbito cumplido**,
    - **coste cumplido**,
    - *buena calidad*
      - *porque tiene buena mantenibilidad*

Proyecto Software efectivo por buena disciplina/metodología



- Producto Software efectivo
  - porque tiene **buena calidad**
    - Es **eficiente**
    - Es eficaz en **corrección**, **usabilidad**, **escalabilidad**, ...
    - porque tiene **buena mantenibilidad**, de la que depende la eficiencia y la eficacia anteriores, porque es
      - **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



- **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 tremendamente 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; oxigeno 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** per se utilizados 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

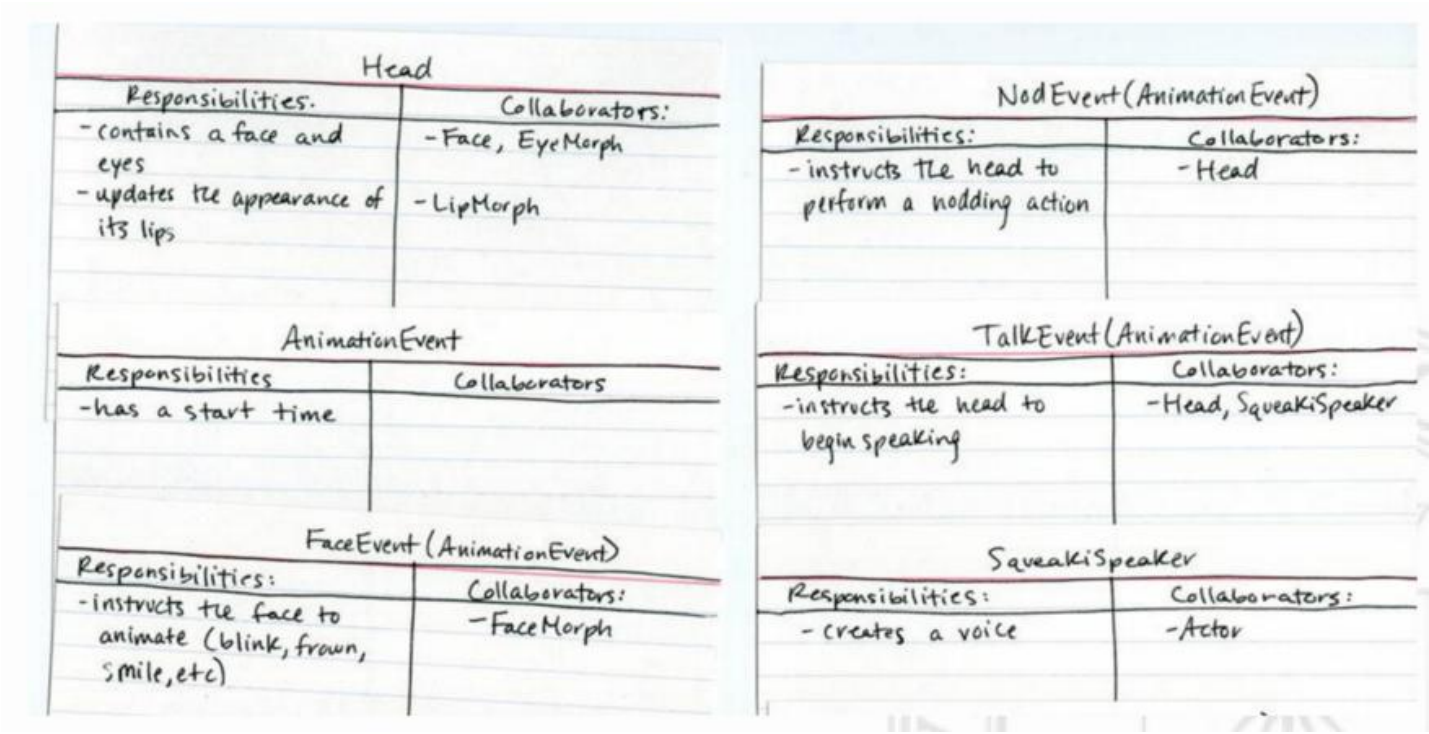
- Un intento para identificar los **objetos, las operaciones y las relaciones [son los que] los expertos de dominio perciben** como importantes sobre el dominio.
  - A menudo, un experto de dominio es **simplemente un usuario**, *como un ingeniero del tren o expendedor en un sistema ferroviario, o una 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-responasability-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



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 contenedor**. 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*

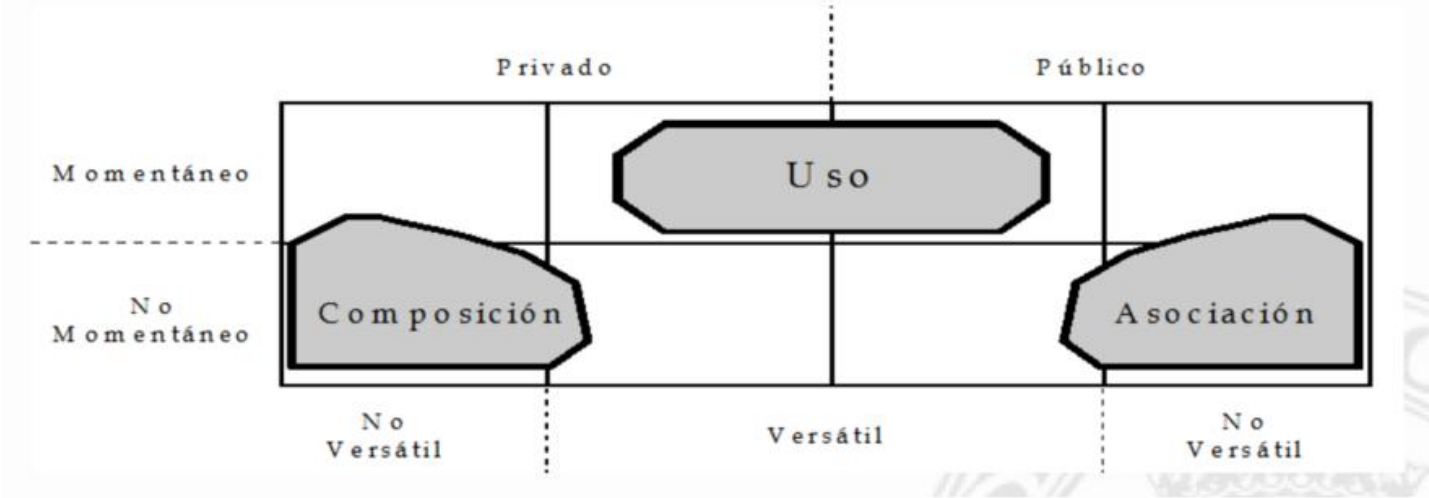
Relación de Asociación

- 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 automovilísticos (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	Antipatrónde Desarrollo	

- **Síntomas**: clases con nombresde 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/IWVG/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 permanencer juntas
  - Las variables deberían declararse tan cerca comos sea posible de su utilización, **minimizar el intervalo de vida de una variable**
  - Los programadores prefieren **líneas cortas** (~40, máximo80/120)
  - 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.: *intdayOfMonth; //the day of themonth,*
  - 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 'l' 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

- **Codigo 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 convencion 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 Chernobylse 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**:
  - Según el código muerto se anquilosa y se endurecen, rápidamente se hace **imposible documentar** el código o entender suficientemente su arquitectura para hacer mejoras.
  - Si no se elimina el código muerto, **puede continuar proliferando** según se reutiliza código en otras áreas

- Puede haber crecimiento exponencial según los sucesivos desarrolladores, demasiado apremiados o intimidados por analizar los códigos originales, seguirán produciendo **nuevos flujos secundarios** en su intento de evitar los originales.
- 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	Antipatrónde 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 consecuenciaes que una modificación de cualquier elemento individual de un sistema **no requiere un cambio en otros elementos** lógicamente no relacionados (similar a la 1ªFN de BBDD).
  - **Aplicable a 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*