POLITÉCNICA
"Ingeniamos el futuro"

CAMPUS
DE EXCELENCIA
INTERNACIONAL

MiW

Visión General de la Ingeniería Web.

4. Análisis

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

INDICE

1. Introducción
2. Enfoques de Clasificación
3. Clases de Análisis
4. Relaciones entre Clases
5. Estrategias de Análisis

4.1. Introducción

- 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
  - Acercase 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

4.1. Introducción

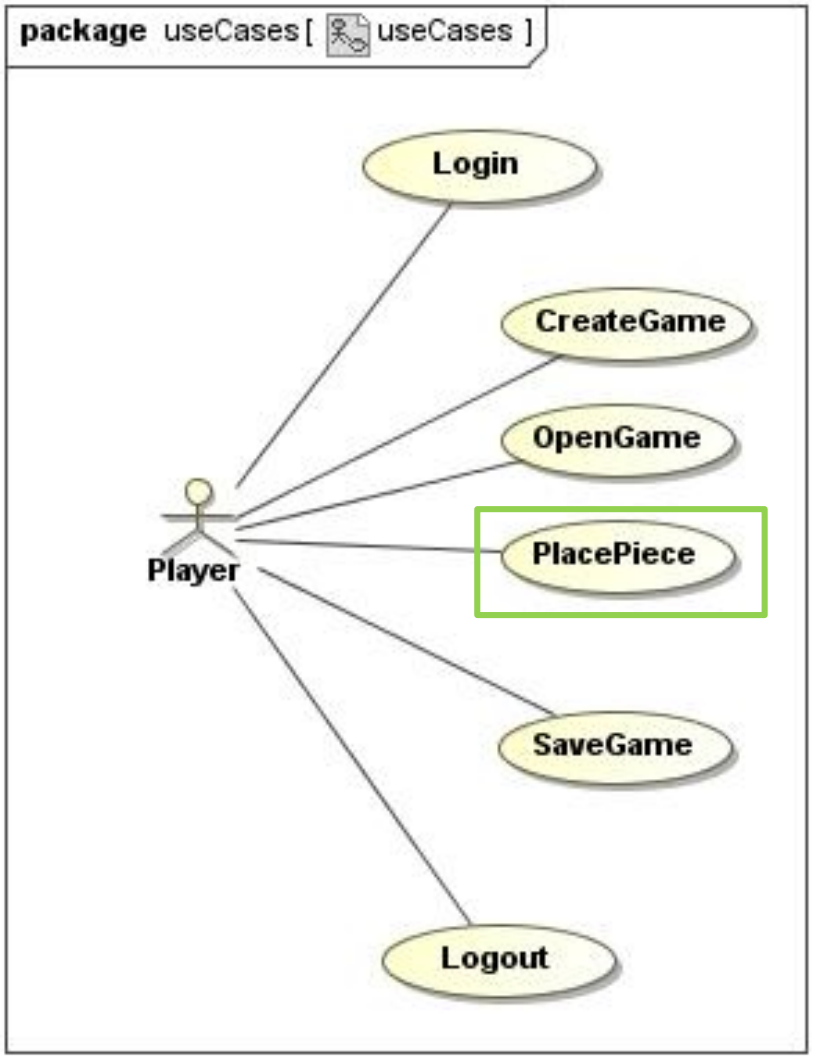
- Comparativa entre requisitos y análisis:
- **Requisitos:**
  - Descrito usando el lenguaje del cliente
  - Visión externa del sistema
  - Estructurado por requisitos, da estructura a la vista externa
  - Usado principalmente como contrato entre los clientes y los desarrolladores sobre lo que el sistema debería hacer
- **Análisis:**
  - Descrito usando el lenguaje de los desarrolladores
  - Visión interna del sistema
  - Estructurado por clases estereotipadas y paquetes, da estructura a la vista interna
  - Usado principalmente por desarrolladores para comprender qué forma debería tener el sistema (p.ej. diseño e implementación)

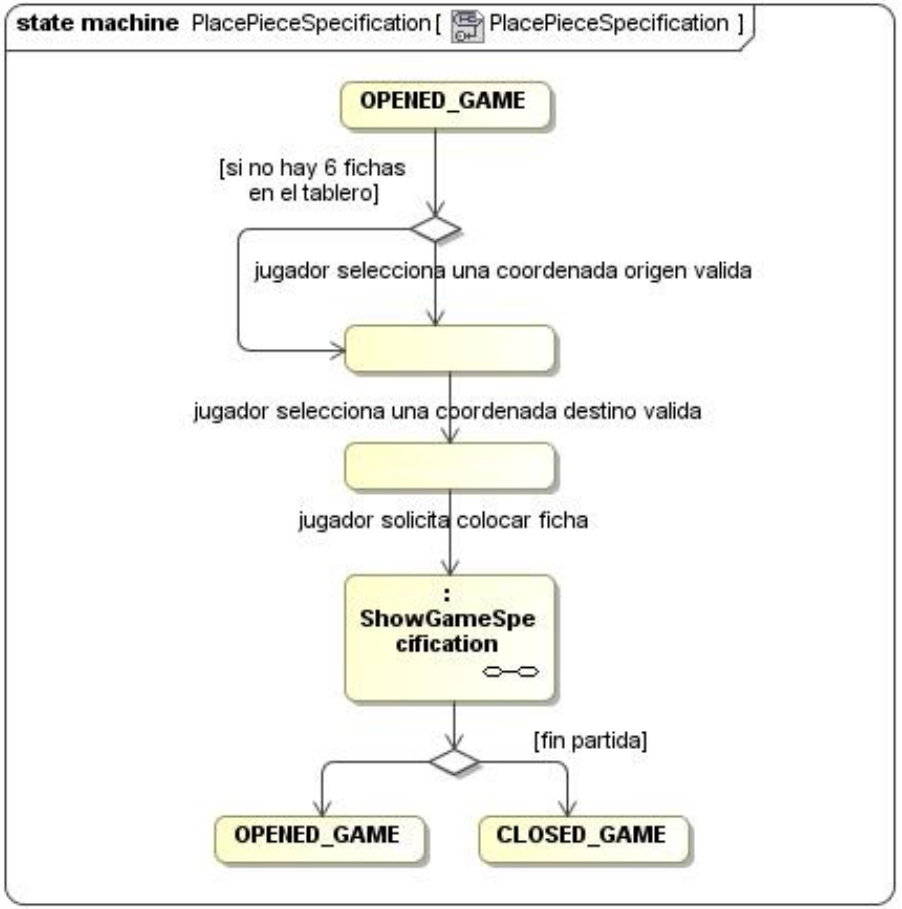
4.1. Introducción

- Comparativa entre requisitos y análisis:
- **Requisitos:**
  - Contiene muchas redundancia, inconsistencias, .. entre los requisitos
  - Captura la funcionalidad del sistema, incluyendo funcionalidad arquitectónica significativa
- **Análisis:**
  - No debería contener redundancias, inconsistencias, … entre los requisitos
  - Esboza cómo realizar la funcionalidad en el sistema, incluyendo la funcionalidad arquitectónica significativa; funciona como un primer corte del diseño

4.1. Introducción

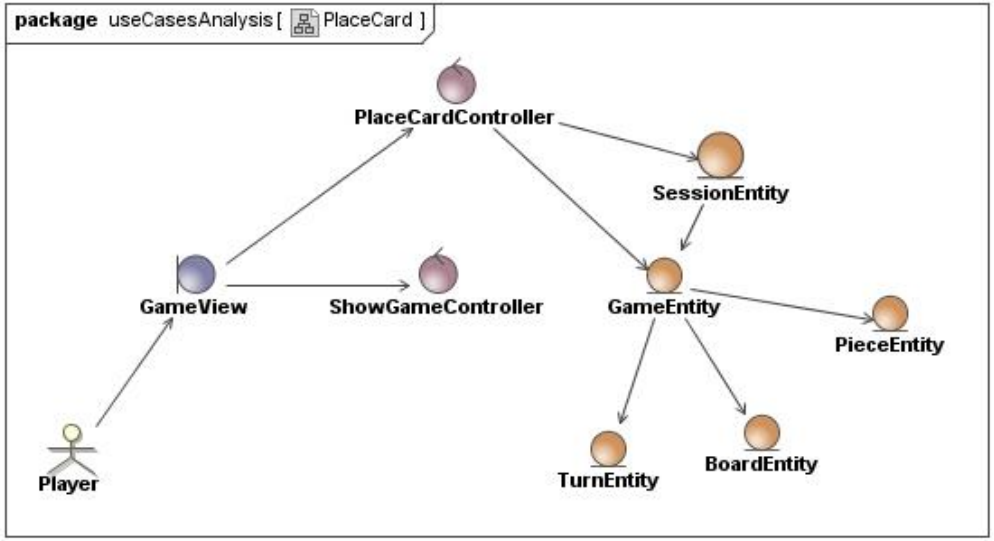
- Salida de requisitos y entrada al análisis:

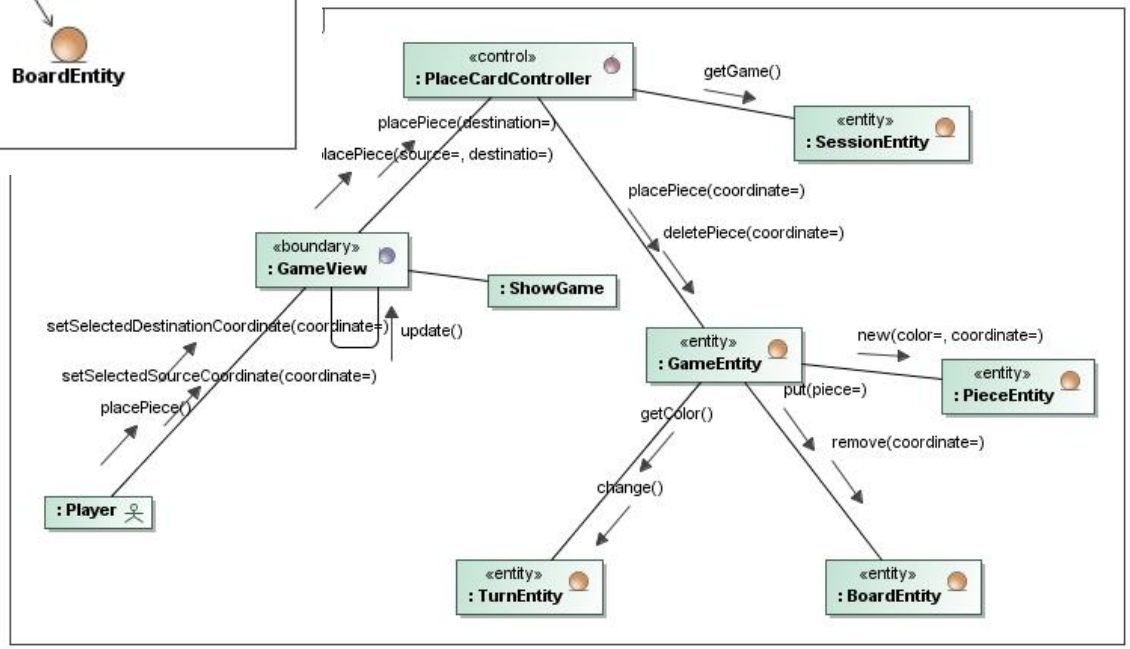




4.1. Introducción

- Salida del análisis y entrada al diseño:





4.1. Introducción

- Comparativa entre análisis y diseño:
- **Análisis:**
  - Modelo conceptual porque es una abstracción del sistema y evita cuestiones de implementación
  - Menos formal
  - Diseño genérico, aplicable a varios diseños concretos
  - Tres estereotipos conceptuales en las clases: modelo, vista, controlador
- **Diseño:**
  - Modelo físico porque es un esbozo de la implementación
  - Más formal
  - No es genérico sino específico para una implementación
  - Cualquier número de estereotipos físicos en las clases, dependiendo del lenguaje de implementación

4.1. Introducción

- Comparativa entre análisis y diseño:
- **Análisis**:
  - Menos costoso para el desarrollo (1:5 frente al diseño)
  - Pocas capas arquitectónicas
  - Puede no ser mantenido a través de todo el ciclo de vida del software
  - Principalmente creado en trabajo de campo, talleres y similares
- **Diseño**:
  - Más costoso para el desarrollo (5:1 frente al análisis)
  - Muchas capas arquitectónicas
  - Debería ser mantenido a través de todo el ciclo de vida del software
  - Principalmente creado por "programación visual" (ingeniería directa e inversa)

4.1. Introducción

- Comparativa entre análisis y diseño:
- **Análisis**:
  - Define la estructura que es la entrada esencial para dar forma al sistema, incluyendo la creación del modelo de diseño
  - Enfatiza la investigación del problema y sus requisitos
  - **Haz lo correcto**
- **Diseño**:
  - Dar forma al sistema mientras intenta preservar la estructura definida por el modelo de análisis
  - Enfatiza en la solución conceptual que cubra los requisitos más que en su implementación
  - **Hazlo correctamente**

2. Enfoques de Clasificación

1. Categorización clásica
2. Agrupación conceptual
3. Teoría de Prototipos

4.2.1. Categorización clásica

- Definición: "*Todas las entidades que tienen una determinada propiedad o conjunto de propiedades en común forman una categoría.* ***Estas propiedades son necesarias y suficientes para definir la categoría***". **[Booch]**
  - Implicación: "*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*" **[Kosko]**

4.2.2. Agrupación conceptual

- Defincción: *"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 de un agrupación probabilística de objetos

4.2.3. Teoría de Prototipos

- Defincción: "*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**]
- Implicación:
  - 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. Nosotros agrupamos cosas según el grado de su relación con prototipos concretos.

3. Clases de Análisis

1. Introducción
2. Experto en Información
3. Alta Cohesión
4. Bajo Acoplamiento
5. Invención Pura
6. Controlador
7. Creador
8. Indirección
9. Polimorfismo
10. Variaciones Protegidas

4.3.1. Introducción

- GRASP **[Craig Larman]:**
  - Principios o Patrones Generales de Software para Asignación de Responsabilidades:
    - *General Responsibility Assignment Software Patterns or Principles*
    - *Grasp* a su vez significa: conocimiento, entendimiento, … comprender, entender
  - Ayudan al aprendizaje de comprender el diseño objeto esencial, y aplicar el razonamiento del diseño de una manera metódica, racional, explicable.
  - Este enfoque de comprensión y uso de los principios de diseño se basa en los Patrones de Asignación de Responsabilidades

4.3.2. Experto en la Información

- Problema. **¿Cuál es un principio general de la asignación de responsabilidades a los objetos?**
  - Un modelo de diseño puede definir cientos o miles de clases de software y una aplicación puede requerir cientos o miles de responsabilidades que debe cumplir. Durante el diseño de objetos, cuando se definen las interacciones entre los objetos, tomamos decisiones sobre la asignación de responsabilidades a las clases de software. Hecho así, los sistemas tienden a ser más fáciles de entender, mantener y ampliar, y hay más oportunidad de reutilizar componentes en aplicaciones futuras.
- Solución. Asignar la responsabilidad al experto en la información: **la clase que tiene la información necesaria para cumplir con la responsabilidad**

4.3.2. Experto en la Información

- Discusión. [**Larman**]
  - Experto en la Información se utiliza con frecuencia en la asignación de responsabilidades; se trata de un **principio rector básico que se utiliza de forma continua en el diseño de objetos**. Expresa la "intuición" común de que los objetos hacen cosas relacionadas con la información que tienen.
  - El patrón de Experto en la Información, como muchas cosas en la tecnología de objetos, tiene una **analogía en el mundo real**. Comúnmente damos la responsabilidad a las personas que tienen la información necesaria para cumplir con una tarea.
  - El Experto generalmente 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.
  - 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**.

4.3.2. Experto en la Información

- Beneficios:
  - La encapsulación de la información se mantiene, ya que los objetos utilizan su propia información para cumplir con las tareas. Esto por lo general **apoya el bajo acoplamiento**, lo que conduce a sistemas más robustos y fáciles de mantener.
  - Comportamiento se distribuye a través de las clases que tienen la información necesaria, fomentando así las **definiciones más cohesionadas de clases** que son más fáciles de entender y mantener.
- Contraindicaciones:
  - 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 arquitectura: diseño con una 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**

4.3.3. Alta Cohesión

- Problema: **¿Cómo mantener la complejidad manejable?**
  - La cohesión, o más específicamente, la cohesión funcional, es una medida de cómo de fuerte están relacionadas y enfocadas las responsabilidades que son de un elemento.
  - Un elemento con responsabilidades muy relacionadas y que no hace una enorme cantidad de trabajo, tiene una alta cohesión. Estos elementos incluyen clases, subsistemas, y así sucesivamente.
  - Una clase con baja cohesión hace muchas cosas no relacionadas, o hace demasiado trabajo. Clases de baja cohesión a menudo representan un "grano muy grande" de abstracción o han asumido responsabilidades que deberían haber sido delegadas a otros objetos.
  - Estas clases son indeseables; adolecen de los siguientes problemas: difícil de comprender, de reutilizar, de mantener y delicada (constantemente afectada por el cambio)
- Solución: **Asignar una responsabilidad para que la cohesión sigue siendo alta**

4.3.3. Alta Cohesión

- Discusión:
  - Alta cohesión es un principio a tener en cuenta en todas las decisiones de diseño; **es un objetivo fundamental para considerar continuamente**. Es un principio valorativo que un diseñador aplica al evaluar todas las decisiones de diseño.
  - El patrón de alta cohesión como muchas cosas en la tecnología de objetos tiene una **analogía del mundo real**. Es una observación común que si una persona toma demasiadas responsabilidades no relacionadas, en especial las que adecuadamente se deben delegar a otros, entonces la persona no es eficaz.

4.3.3. Alta Cohesión

- Discusión:
  - Escenarios que ilustran diversos grados de cohesión funcional:
    - **Muy baja cohesión**. Una clase es la única responsable de muchas cosas en muy diferentes áreas funcionales.
    - **Baja cohesión**. Una clase tiene una responsabilidad exclusiva de una tarea compleja en un área funcional.
    - **Cohesión moderada**. Una clase tiene responsabilidades ligeras y únicas en unas pocas áreas diferentes que están lógicamente relacionadas con el concepto de la clase, pero no entre sí.
    - **Alta cohesión**. Una clase tiene responsabilidades moderadas en un área funcional y colabora con otras clases para cumplir con las tareas. Como regla general, una clase con alta cohesión tiene un número relativamente pequeño de métodos, con una funcionalidad muy relacionadas, y no hace demasiado trabajo. Colabora con otros objetos para compartir el esfuerzo si la tarea es grande.

4.3.3. Alta Cohesión

- Contraindicaciones. Hay unos pocos casos en los que se justifica la aceptación de una cohesión menor
  - Uno de los casos es la agrupación de responsabilidades o código en una clase o componente para simplificar el mantenimiento de una persona aunque se advierte que tal agrupación también puede hacer el mantenimiento peor: **Clases de Utilidad**
  - Otro caso para los componentes con menor cohesión es con objetos de servidores distribuidos. Debido a las implicaciones generales y de rendimiento asociados con objetos remotos y la comunicación a distancia, a veces es deseable para crear menos y más grandes objetos servidores menos cohesivos que proporcionan una interfaz para muchas operaciones.
- Beneficios:
  - Se aumenta la claridad y facilidad de comprensión del diseño.
  - Se simplifican mantenimiento y mejoras .
  - Alta Cohesión es a menudo compatible con Bajo Acoplamiento
  - El grano fino de la funcionalidad altamente relacionada soporta una mayor reutilización porque una clase cohesiva se puede utilizar para un propósito muy específico

4.3.4. Bajo Acoplamiento

- Problema. **¿Cómo apoyar el bajo acoplamiento, el bajo impacto del cambio y el aumento de la reutilización?**
  - El acoplamiento es una medida de la fuerza con un elemento está conectado a, tiene conocimiento de, o se basa en otros elementos. Un elemento con bajo (o débil) acoplamiento no es dependiente de muchos otros elementos; "demasiados" es dependiente del contexto, pero se examinará. Estos elementos incluyen clases, subsistemas, sistemas, y así sucesivamente.
  - Estas clases pueden ser indeseables; algunas sufren los siguientes problemas:
    - Los cambios en las clases relacionadas fuerzan cambios locales.
    - Más difícil de entender de manera aislada.
    - Más difícil de reutilizar porque su uso requiere la presencia adicional de las clases de las que es dependiente.
- Solución. **Asignar responsabilidades de tal manera que el acoplamiento siga siendo bajo**

4.3.4. Bajo Acoplamiento

- Discusión:
  - Formas de **acoplamiento directo** de un *Tipo X* a un *Tipo Y* incluyen:
    - *Tipo X* tiene un atributo que hace referencia a una instancia *Tipo Y*.
    - *Tipo X* tiene un método que hace referencia a una instancia de *Tipo Y*, por cualquier medio. Estos suelen incluir un parámetro o variable local de tipo *Tipo Y*, o el objeto de retorno de un mensaje es una instancia de *Tipo Y*.
    - Un objeto *Tipo X* pide a los servicios de un objeto *Tipo Y*.
    - *Tipo X* es una clase derivada directa o indirecta de *Tipo Y*. Existe una **tensión entre los conceptos de acoplamiento y herencia:** las clases débilmente acopladas son deseables; y la herencia, la cual acopla parejas de superclases y sus subclases, nos ayuda a explotar el factor común entre las abstracciones.
  - Forma de **acoplamiento indirecto** de un *Tipo X* a un *Tipo Z* será cuando el primero envía mensajes a objetos de *Tipo Z* que devuelvan los objetos de *Tipo Y* por un acoplamiento directo

4.3.4. Bajo Acoplamiento

- Discusión:
  - No hay una medida absoluta de acoplamiento cuando es demasiado alto. Lo que es importante es que **un desarrollador puede medir el grado actual de acoplamiento, y evaluar si el aumento dará lugar a problemas.** En general, las clases que son inherentemente de naturaleza muy genéricas, y con una alta probabilidad para su reutilización, deberían tener especialmente bajo acoplamiento.
  - El Bajo Acoplamiento alienta **la asignación de una responsabilidad de forma que su colocación no aumente el acoplamiento a un nivel tal que conduce a los resultados negativos** que el alto acoplamiento puede producir.
  - **Un grado moderado de acoplamiento entre clases es normal y necesario** para la creación de un sistema orientado a objetos en el que las tareas se cumplen por una colaboración entre los objetos conectados

4.3.4. Bajo Acoplamiento

- Discusión:
  - El Bajo Acoplamiento es un principio a tener en cuenta en todas las decisiones de diseño; es un **objetivo fundamental para considerar continuamente**.
  - En la práctica, el nivel de acoplamiento por sí solo **no puede considerarse en forma aislada de otros principios como el Experto y Alta Cohesión**. Sin embargo, es un factor a considerar en la mejora de un diseño.
  - **El alto acoplamiento per se no es el problema; es el acoplamiento a elementos que son inestables en alguna dimensión, como su interfaz, su implementación, o su mera presencia**. Podemos añadir flexibilidad, encapsular datos e implementaciones y diseñar en general con bajo acoplamiento en muchas áreas del sistema . Pero si ponemos esfuerzos en “posibles futuros" o bajando el acoplamiento en algún punto en el que, de hecho, no hay motivación realista, esto no será un tiempo bien empleado. Los diseñadores tienen que elegir sus batallas en la reducción de acoplamiento y centrarse en los puntos de alta inestabilidad o evolución realista.

4.3.4. Bajo Acoplamiento

- Contraindicaciones:
  - **Alto Acoplamiento a elementos estables y generalizados rara vez es un problema**. Por ejemplo, una aplicación Java J2EE de forma segura se puede acoplar a las bibliotecas de Java (java.util, y así sucesivamente), porque son estables y generalizadas
- Beneficios:
  - **No se ve afectado** por los cambios en otros componentes
  - **Fácil de entender** de manera aislada
  - Conveniente para **reutilizar**

4.3.5. Invención Pura

- Problema: **Hay muchas situaciones en las que la asignación de responsabilidades solamente a las clases de software de la capa de dominio da lugar a problemas en cuanto a la escasa cohesión, elevado acoplamiento o de bajo potencial de reutilización**
- Solución: **Asignar un conjunto altamente cohesivo de responsabilidades a una clase artificial o de conveniencia que no representa un concepto (algo dominio del problema) , para apoyar alta cohesión, bajo acoplamiento, y reutilización**
  - Ejemplo: "cadena de caracteres poética" con responsabilidades sobre las rimas que no se desean asignar a la "cadena de caracteres" general; "formateador de intervalos" para distintas presentaciones; …

4.3.6. Controlador

- Problema: **¿Quién debe ser responsable de manejar un evento de sistema de entrada generado por un agente externo?**
  - Un evento de entrada al sistema es un evento generado por algún actor externo – personas mediante el teclado o el ratón - o máquinas mediante señales de un sensor o tramas de red-.
- Solución: **Asignar la responsabilidad de recibir o manejar un mensaje de eventos del sistema a una clase que representa una de las siguientes opciones:**
  - Representa el sistema, dispositivo o subsistema en general.
  - Representa un caso de uso dentro del cual se produce el evento del sistema, a menudo llamado <*UseCaseName*>*Controller*. Utilice la misma clase controlador para todos los eventos del sistema del mismo caso de uso.
  - Son objetos asociados con operaciones del sistema como mensajes, métodos. Un controlador es un objeto responsable de recibir o manejar los eventos del sistema. Los controladores definen los métodos para las operaciones del sistema.
  - Las clases de la interfaz no deberían contener tareas asociadas con los eventos, típicamente los reciben y delegan a un controlador

4.3.6. Controlador

- Implicaciones:
  - Informalmente, **una sesión es una instancia de una conversación con un actor**. Las sesiones pueden ser de cualquier longitud, pero se organizan a menudo en términos de casos de uso.
  - Un **controlador no es un objeto de interfaz de usuario** responsable de recibir o manejar un evento del sistema. Un controlador define el método para la operación del sistema asociada al evento. Tenga en cuenta que clases “Window", "Widget "," View" y “Document" no están en esta lista: estas clases no deben cumplir con las tareas asociadas a los eventos del sistema, suelen recibir estos eventos y delegarlos a un controlador.
  - Normalmente, un controlador debe **delegar a otros objetos el trabajo que hay que hacer; coordina o controla la actividad**. No hace mucho el trabajo en sí.

4.3.7. Creador

- Problema: **¿Quién debería ser el responsable de crear un nueva instancia de alguna clase?**
  - La creación de objetos es una de las actividades más comunes en un sistema orientado a objetos. Consecuentenemente, es útil tener un principio general para asignar la responsabilidad de creación. Con buenas asignaciones, el diseño puede soportar bajo acoplamiento, incrementar la claridad, encapsulación y reusabilidad
- Solución: **Asignar a la clase B la responsabilidad de crear instancias de la clase A, B es un creador de objetos de A, si uno o más de las siguientes condiciones es cierta:**
  - B contiene/agrega objetos de A
  - B registra instancias de objetos de A
  - B usa estrechamente objetos de A
  - B tiene los datos de inicialización a ser pasados cuando es creado A (entonces B es un experto en la creación de A)

4.3.7. Creador

- Discusión:
  - La intención básica del patrón Creador es encontrar un creador que necesita estar conectado al objeto creado en cualquier situación. Eligiendolo como un Creador se soporta el bajo acoplamiento.
  - A menudo, la creación requiere una complejidad significativa, como reciclar instancias por razones de rendimiento, creación condicional de una instancia de una familia de clases similares basándose en el valor de alguna propiedad externa, etc.
- Beneficios:
  - Bajo acoplamiento es compatible, lo que implica dependencias de mantenimiento más bajos y mayores oportunidades para la reutilización. El acoplamiento probablemente no se incrementa porque la clase creada es probable que ya sea visible a la clase Creador, debido a las asociaciones existentes que motivaron su elección como Creador

4.3.8. Indirección

- Problema. **¿Dónde asignar una responsabilidad para evitar el acoplamiento directo entre dos (o más) cosas? ¿Cómo desacoplar objetos para que se fomente el bajo acoplamiento y que el potencial de reutilización siga siendo el más elevado?**
- Solution. **Asignar la responsabilidad a un componente intermedio para mediar entre otros componentes o servicios para que no se acoplen directamente. El intermediario crea una indirección entre los otros componentes.**

La mayoría de los problemas en las Ciencias de la Computación pueden ser resueltos con otro nivel de indirección!

La mayoría de los problemas de rendimiento pueden ser resueltos eliminando otro nivel de indirección

[Dennis DeBruler]

4.3.8. Indirección

- Implicaciones:
  - Permitir compartir la lógica. Un sub-método invocado en dos lugares diferentes o un método en una clase base compartida por todos las derivadas
  - Explicar la intención e implementar separadamente. Elegir los nombres de cada clase y cada método da la oportunidad de explicar las intenciones. En el interior de una clase o método explican cómo las intenciones se realizan. Si el interior también se escribe en términos de intencion con piezas todavía más pequeñas, se puede escribir código que comunica la mayoría de la información importante sobre su estructura
  - Aislar cambios. Si se una un objeto en dos lugares, se quiere cambiar el comportamiento en uno de ellos y cambiar el objeto produce cambios en los dos, se puede hacer una subclase y referirse a ella en el caso del cambio. Se puede modificar la clase sin arriesgarse a cambiar inadvertidamente a cambiar otros casos

4.3.8. Indirección

- Implicaciones:
  - Codificar la lógica condicional. Los objetos tienen un mecanismo fabuloso, paso de mensajes polimórficos, para, de forma flexible pero clara, expresar lógica condicional. Al cambiar explícitamente los condicionales por mensajes, se puede reducir la duplicación, añadir claridad e incrementar la flexibilidad al mismo tiempo
- Contraindicaciones:
  - Si se encuentran métodos que se usaron para servir un propósito pero no llegó a llevarse a cabo; o si se encuentra un componente que se esperaba compartir o ser polimórfico pero sólo se usa en un sitio; … cuando se encuentre indirecciones parasitarias, quítalas.

4.3.9. Polimorfismo

- Problema: **¿Cómo manejar alternativas basadas en un tipo? ¿Cómo crear componentes software conectables (pluggin)?**
  - Alternativas basadas en tipos. Las alternativas condicionales es un tema fundamental en los programas. Si un programa está diseñado usando sentencias *if-then-else* o *switch*, entonces si nuevas alternativas aparecen, requiere la modificación de la lógica. Este enfoque hace difícil extender fácilmente un programa con una nueva alternative porque los cambios tienden a realizarse en diferentes lugares
  - Componentes software conectables. Viendo componentes en una relación cliente/servidor, ¿cómo puedes reemplazar un componente servidor con otro sin afectar al cliente?
- Solución: **Cuando alternativas o comportamientos relacionados varían por tipo (clase), asignar la responsabilidad del comportamiento, usando operaciones polimórficas, a los tipos en los cuales el comportamiento varía**
  - Colorario: No preguntes por el tipo de un objeto ni uses lógica condicional para realizar variaciones alternativas basadas en un tipo

4.3.9. Polimorfismo

- Contraindicaciones:
  - Algunas veces, los desarrolladores diseñan sistemas con interfaces y polimorfismo especulativamente a prueba de futuro. Si el punto de variación es, sin duda motivado por una variabilidad inmediata o muy probable, entonces el esfuerzo de añadir la flexibilidad a través de polimorfismo es por supuesto racional. Pero se requiere una evaluación crítica, ya que no es raro ver un esfuerzo innecesario que se aplica a prueba de futuro con un diseño de polimorfismo en la variación señalada que, de hecho, son improbables y nunca surgirá en realidad.
  - Sea realista acerca de la verdadera probabilidad de la variabilidad antes de invertir en una mayor flexibilidad.
- Beneficios:
  - Las extensiones requeridas para nuevas variaciones son fáciles de añadir
  - Nuevas implementaciones pueden ser introducidas sin afectar a los clientes

4.3.10. Variaciones Protegidas

- Problema: **¿Cómo diseñar objetos, sub-sistemas o sistemas tal que variaciones o inestabilidades en estos elementos no tenga un indeseable impacto en otros elementos?**
- Solución: **Identificar puntos de variación o inestabilidad previstos y asignar responsabilidades para crear interfaces estables en torno a éstos**
  - El Principio Abierto/Cerrado de **Meyer** (SOLID) y este patrón son esencialmente dos expresiones del mismo principio con diferente énfasis: puntos de evolución y variación
  - La ocultación de información de **Parnas** es el mismo principio y no simplemente la encapsulación de datos, la cual es una de las muchas técnicas de ocultación de la información en el diseño. Sin embargo, el término ha sido ampliamente re-interpretado como un sinónimo de encapsulación de datos y ya no es posible utilizarlo en su sentido original sin malentendido. Éste incluye, ocultar la clase contreta a través de una interfaz, …
  - Proponemos que comenzar con una lista de las decisiones de diseño difíciles o de decisiones de diseño que puedan cambiar. Entonces, cada módulo es diseñado para ocultar tal decisión a los demás

4.3.10. Variaciones Protegidas

- Implicaciones:
  - El patrón de Variaciones Protegidas es la raíz principal de la motivación de la mayoría de los mecanismos y patrones en programación y diseño para proveer flexibilidad y protección desde las variaciones.
  - Mecanismos nucleares de Variaciones Protegidas. La encapsulación de datos, el polimorfismo, la indirección y los estándares son motivados por el patron de Variaciones Protegidas. Darse cuenta de que comoponentes como brokers o máquinas virtuales son ejemplos complejos de indirección para realizar el patron de Variaciones Protegidas
  - Diseños dirigidos por datos cubren una amplia familia de técnicas que incluyen lectura desde una fuente externa de códigos, valores, nombres de fichero *.class*, nombres de clases, … para cambiar el comportamiento o parametrizar un sistema de alguna forma en tiempo de ejecución. Otras variaciones incluyen hojas de estilo, metadatos para la proyección objeto-relacional, ficheros de propiedades, …

4.3.10. Variaciones Protegidas

- Contraindicaciones:
  - Es útil definer dos tipos de puntos de cambio:
    - Punto de variación donde los requisitos o el sistema actual deben ser soportados
    - Puntos de evolución que pueden erigirse en el future pero no están presentes en los requisitos actuales
  - Algunas veces el coste de pruebas a futuro especulativas en un punto de evolución supera el costo incurrido por un diseño más simple y más "frágil" que el diseño necesario en respuesta a las verdaderas presiones del cambio. Es decir, el coste de la protección de ingeniería en los puntos de la evolución puede ser más alto que volver a trabajar sobre un diseño simple.
  - El equilibrio no es abogar por el re-trabajo y diseños frágiles. Si la necesidad de flexibilidad y la protección contra el cambio es realista aplicar Variaciones Protegidas está motivado. Pero si es para prueba a futuro o "reutilización" especulativa con probabilidades muy inciertas, entonces se llama a la moderación y el pensamiento crítico.

4. Relaciones entre Clases

1. Tipos de Relaciones entre Clases
2. Características de las Relaciones entre Clases por Colaboración
3. Relación de Composición/Agregación
4. Relación de Asociación
5. Relación de Dependencia (uso)
6. Comparativa de las Relaciones entre Clases por Colaboración
7. Relación de Herencia

4.4.1. Tipos de Relaciones entre Clases

- **Colaboración entre Objetos**: si un objeto envía mensajes a otro
- **Relación entre Clases**:
  - si dos objetos de sus respectivas clases colaboran. Tipos de relación:
    - **Relación de Composición/Agregación**
    - **Relación de Asociación**
    - **Relación de Dependencia (uso)**
  - 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**
- **Dependencia** es la nueva forma de referirse a una relación entre clases:
  - La clase del objeto que envía mensajes al objeto de la otra clase, depende de ésta última
  - La clase hija en una relación de herencia depende de la clase padre

4.4.2. Características de las Relaciones entre Clases por Colaboración

- **Visibilidad**: carácter privado o público de la colaboración entre dos objetos.
  - Por ejemplo: 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.
  - Por ejemplo: 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.
  - Por ejemplo: un profesor colabora con su computadora para preparar la documentación de un curso y colabora con cualquier computadora para consultar el correo electrónico.

4.4.3. Relación de Composición/Agregación

- **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

4.4.3. Relación de Composición/Agregación

- 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

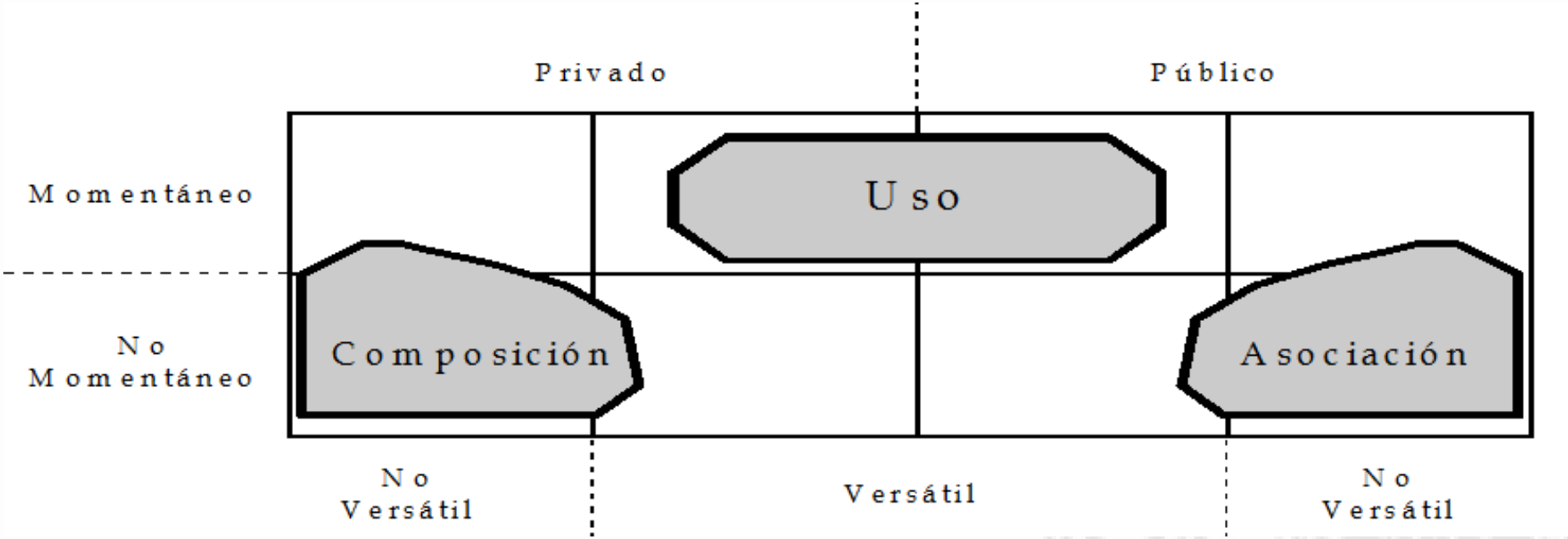
4.4.4. Relación de Asociación

- **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

4.4.5. Relación de Dependencia (Uso)

- **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

4.4.6. Comparativa de las Relaciones entre Clases por Colaboración



- La **agregación** sería un área difusa entre:
  - la composición por la izquierda
  - la asociación por la derecha

4.4.6. Comparativa de las Relaciones entre Clases por Colaboración

- 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

4.4.6. Comparativa de las Relaciones entre Clases por Colaboración

- *"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**, 91]
  - **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.

4.4.6. Comparativa de las Relaciones entre Clases por Colaboración

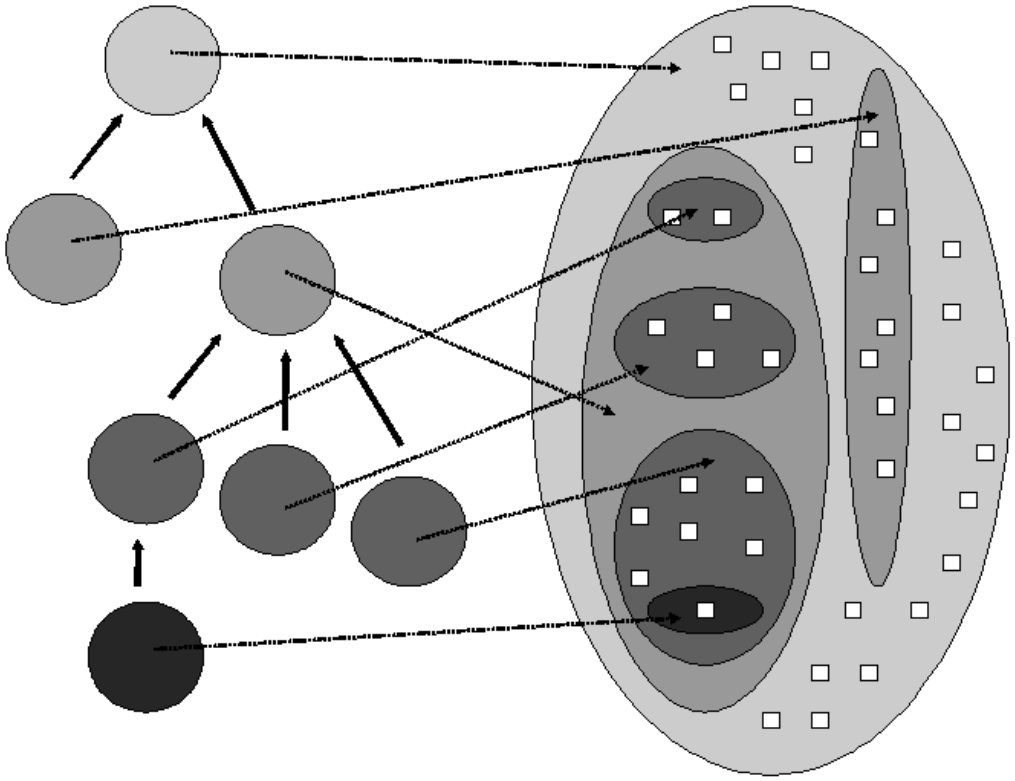
- El **objetivo** principal de establecer relaciones entre clases es:
  - Analizar/diseñar la forma en qué colaboran los objetos para llevar a cabo los requisitos del sistema, siendo secundario, e imposible, determinar a qué relación responden exactamente en todas las ocasiones; Analizar/diseñar entre qué clases no existe relación.
- Al igual que no existe una fórmula para determinar cuál es la relación dada en una colaboración entre objetos, no existen fórmulas para **traducir la relación escogida a un código particular**.
  - Por tanto, sólo se establecerán pautas de actuación que ayudarán al programador a formar un esqueleto del programa (no todos los detalles) en un amplio abanico de casos pero será responsabilidad última del programador saber cuándo romper las reglas:

4.4.6. Comparativa de las Relaciones entre Clases por Colaboración

- Pautas de traducción de la Relación entre Clases a la Implementación:
  - La **relación de composición**: la clase "todo" crea un objeto de la clase "parte" que mantiene como una referencia atributo
  - La **relación de agregación**: la clase "todo" mantiene una referencia atributo a un objeto de la clase "parte" que se suministra desde un constructor o método para añadir el agregado
  - La **relación de asociación**: la clase "cliente" mantiene una referencia atributo a un objeto de la clase "servidor" que se suministra desde un constructor o método para establecer la asociación
  - La **relación de dependencia/uso**: la clase "cliente" contempla como parámetro o valor devuelto de un método un objeto de la clase "servidor" si la colaboración es pública; mientras que contempla como objeto local de un método si la colaboración es privada

4.4.7. Relación de Herencia

- **Relación de Herencia:** es la transmisión de la vista pública (métodos públicos) y de la vista privada (atributos, métodos privados y definición de los métodos) de una clase a otra.



4.4.7. Relación de Herencia

- Reglas de Construcción de la Relación de Herencia:
  - **ISA** (acrónimo de *¿… is a …?* - ¿… es un …?): responder afirmativamente a la pregunta de. ¿<un elemento del dominio del nodo hijo> es un <elemento del dominio del nodo padre>?
  - **Generalización/Especialización** es la presencia de unas características específicas de un subconjunto de elementos de un determinado conjunto como, para que si bien mantienen las características esenciales e identificativas del conjunto al que pertenecen, también son lo suficientemente relevantes como para ser rasgos distintivos de dicho subconjunto de elementos

5. Estrategias de Clasificación

1. Descripción informal
2. Análisis clásico
3. Análisis del Dominio
4. Análisis del Comportamiento
5. Análisis de Casos de Uso

4.5.1. Descripción informal

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

4.5.2. Análisis clásico

- Estrategia: 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

4.5.2. Análisis clásico

- Estrategia **fuentes de clases y objetos** :
  - 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

4.5.3. Análisis del Dominio

- Estrategia: 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.

4.5.4. Análisis del Comportamiento

- Estrategia: 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**]

4.5.4. Análisis del Comportamiento

- **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.
    - 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

4.5.4. Análisis del Comportamiento

- 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). Esto está superado por las herramientas CASE con UML

4.5.4. Análisis del Comportamiento

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, SqueakiSpeaker

SqueakiSpeaker

Responsibilities:	Collaborators:
- creates a voice	- Actor

4.5.5. Análisis de Casos de Uso

- Estrategia:
  - 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.

4.5.5. Análisis de Casos de Uso

- Estrategia:
  - 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.
  - Podemos aplicar el análisis de casos de uso ya en el análisis de requerimientos, a la que vez que los usuarios finales, otros expertos del dominio y el equipo de desarrollo enumeran los escenarios que son fundamentales para el funcionamiento del sistema.
  - Los escenarios también sirven como la base de las pruebas del sistema.