Análisis y Diseño Orientado a Objetos

Grady Booch

INDICE

Capítulo 1 - Complejidad

1.1 – La complejidad inherente al software

Software de dimensión industrial.

Aplicaciones con un ciclo de vida muy largo y de los cuales muchos usuarios a lo largo del tiempo llegan a depender de su funcionamiento correcto.

Su complejidad excede la capacidad intelectual del ser humano, por lo que resulta imposible ser comprendido en su totalidad por un único desarrollador. Y esto es característica esencial de casi todos los sistemas de software de gran tamaño.

El software es complejo de forma innata. Es una característica esencial de él.

Es complejo porque hereda la complejidad del dominio del problema, la dificultad de gestionar el proceso de desarrollo, la flexibilidad que se puede alcanzar a través del software y los problemas que plantea la caracterización del comportamiento de sistemas discretos.

La complejidad del dominio del problema

1. El problema a resolver presenta muchos requisitos que compiten entre sí o se contradicen (cuando se compite entre **facilidad de uso** o **costo** con la **facilidad de desarrollo** o **su funcionalidad**)
2. Esas contradicciones o desacoplamientos existen debido a que los usuarios suelen encontrar grandes dificultades para expresar sus necesidades en una forma en que los desarrolladores puedan entender. Los usuarios tal vez solo tiene vagas ideas de lo que quieren del sistema.
3. Los requisitos además cambian con frecuencia durante el desarrollo incluso porque la mera existencia de un proyecto de solución lo altera al sistema real.
4. Un sistema grande, debido a la inversión financiera que implica, no puede desecharse y reemplazarse por uno nuevo cada vez que los requisitos cambian. Debe evolucionar.

   Evolucionar del software: responder al cambio de requerimientos

   Mantenimiento del software: corregir errores

   Conservación del software: emplear recursos para mantener en operación un elemento de software anticuado y decadente.

La dificultad de gestionar el proceso de desarrollo

5. La principal tarea del grupo de desarrollo es dar una ilusión de simplicidad para defender a los usuarios de esta complejidad arbitraria del problema.
6. Se hace lo posible por escribir menos código pero a veces es imposible eludir el tamaño de un sistema grande. Debe recurrirse a la aplicación de varias técnicas de re-utilización de código existente o de la escritura de nuevo software.
7. Debe también enfrentarse la existencia de miles de módulos separados y esto implica un grupo de desarrolladores, nunca una única persona. Esto implica más personas y por consiguiente una comunicación más rigurosa y coordinación más difícil, más aún si el grupo y el proyecto se extienden geográficamente.

La flexibilidad que se puede alcanzar a través del software

8. Un proyecto de software es muy frecuentemente apoyado en pilares construidos por los mismos desarrolladores (a diferencia de una obra edilicia, por ejemplo, que no contiene un acería para fabricar sus propias vigas metálicas). Esto significa que el desarrollo del proyecto de software sigue siendo una tarea muy laboriosa. No hay estándares para el desarrollo de software.

Los problemas que plantea la caracterización del comportamiento de sistemas discretos.

9. El **estado actual de una aplicación** está dado por el conjunto de variables (que pueden ser miles), de sus valores actuales y de la dirección de ejecución y de la pila de cada proceso del sistema.
10. En **sistemas continuos** una pequeña modificación en la entrada provoca una pequeña modificación en la salida. Pero en **sistemas discretos** y de **gran tamaño** se percibe una explosión combinatoria que hace que la salida se modifique enormemente.
11. Se intenta diseñar los sistemas con una **separación de intereses**: de forma que el comportamiento de una parte del sistema tenga el mínimo impacto en el comportamiento de otra parte del sistema.
12. En un sistema de gran volumen debe uno contentarse con un **grado de confianza determinado a lo que refiere su corrección** ya que no puede llevarse a cabo una prueba a fondo exhaustiva por no tener las herramientas matemáticas ni intelectuales para un sistema no continuo.

Las consecuencias de la complejidad ilimitada.

Más complejo es el sistema, más abierto está al derrumbamiento total.

Crisis del software: ha existido tanto tiempo que debe considerarse normal. Es cuando se pretende dominar la complejidad del sistema a un extremo que lleva al presupuesto a niveles excedentes y que transforman en deficiente al sistema respecto a los requerimientos fijados.

1.2 La estructura de los sistemas complejos.

Ejemplos de sistemas complejos.

Computador personal: complejidad moderada. Casi todos los computadores están formados por los mismos elementos básicos.

Los sistemas complejos no sólo son jerárquicos, sino que los **niveles de esta jerarquía** representan diferentes niveles de abstracción. A cada **nivel de abstracción** se encuentra una serie de dispositivos que colaboran para proporcionar servicios a capas más altas. Se elige determinado nivel de abstracción para satisfacer **necesidades particulares**.

Sólo a través de la **cooperación mutua de colecciones significativas** de estos agentes se ve la funcionalidad de nivel superior de una planta. la ciencia de la complejidad llama a esto **comportamiento emergente**. El comportamiento del todo es mayor que la suma de sus partes.

Los cinco atributos de un sistema complejo.

1. La complejidad toma la forma de una **jerarquía**, por lo cual un sistema complejo está compuesto de subsistemas relacionados que tiene a su vez sus propios subsistemas, y así sucesivamente, hasta que se alcanza algún ínfimo nivel de componentes elementales.

El hecho de tener una **estructura jerárquica** y casi descomponible es lo que **facilita la comprensión** del sistema complejo. En realidad sólo se puede comprender a los sistemas que tiene una estructura de jerarquía.

2. La elección de qué componentes de un sistema son primitivos es relativamente arbitraria y queda en gran medida a decisión del observador.
3. Los **enlaces internos** de los componentes suelen ser más fuertes que los **enlaces entre los componentes**. Este hecho tiene el efecto de separar la **dinámica de alta frecuencia** de los componentes – que involucra a la estructura interna de los mismos – de la **dinámica de baja frecuencia** – que involucra la interacción entre los componentes.

   Esto posibilita un estudio de cada parte de forma relativamente aislada.
4. Los sistemas jerárquicos están compuestos usualmente de sólo unas pocas clases diferentes de subsistemas dispuestos en varias combinaciones y disposiciones. Esto lleva a la reutilización de componentes pequeños.
5. Se encontrará siempre que un sistema complejo que funciona ha evolucionado de un sistema simple que funcionaba... Un sistema complejo diseñado desde cero nunca funciona y no puede parchearse para conseguir que lo haga. Hay que volver a empezar, partiendo de un sistema simple que funcione.

Complejidad organizada y desorganizada.

La forma canónica de un sistema complejo.

El descubrimiento de abstracciones y mecanismos comunes facilita la comprensión del sistema complejo.

A un sistema se lo puede descomponer en una **jerarquía estructural o «parte-de»** o en una **jerarquía de tipos o «es-un»**. Es esencial ver a un sistema desde ambas perspectivas. A estas jerarquías se las llaman respectivamente **estructura de clases** y **estructura de objetos**.

Por lo general, existen muchos más objetos que clases en un sistema.

Al tener en cuenta la estructura de clases, se capturan las propiedades de determinados objetos en un sólo lugar, posibilitando así una mejor comprensión.

Una nueva mirada a cada nivel de estructura de objetos expone un nuevo nivel de complejidad.

Ambas estructuras no son del todo independientes y juntas conforman la **arquitectura del sistema**.

1.3 - Imponiendo orden al caos.

El rol de la descomposición.

Para estudiar y trabajar con sistemas complejos es muy útil e imperioso dividirlo en partes más y más pequeñas para así poder refinarlas en forma independiente. Divide y vencerás.

Así se satisface la restricción fundamental de la capacidad humana: **para entender un nivel dado basta con comprender unas pocas partes (no necesariamente todas) a la vez**.

Descomposición algorítmica: cada módulo del sistema representa un paso importante de un proceso global. Se siguen las reglas del diseño estructurado y descendente.

Esta visión enfatiza el orden de los eventos.

Descomposición orientada a objetos: se identifican los objetos que se derivan directamente del vocabulario del dominio del problema. Son agentes autónomos que colaboran para llevar acabo algún comportamiento de nivel superior. Un objeto aquí es una entidad tangible que muestra un comportamiento bien definido.

Esta visión resalta a los objetos que provocan acciones o son sujetos de estas acciones.

Ambas visiones son importantes, pero no se puede diseñar un sistema complejo con las dos visiones al mismo tiempo, ya que son vistas perpendiculares.

Hay que comenzar a descomponer un sistema por algoritmos o por objetos y luego utilizar la estructura resultante como marco de referencia para expresar la otra perspectiva.

Categorías de métodos de diseño.

Método es un proceso disciplinado para generar un conjunto de modelos que describen varios aspectos de un sistema de software en desarrollo, utilizando alguna notación bien definida.

Metodología es una colección de métodos aplicados a lo largo del ciclo de vida del desarrollo del software y unificados por alguna aproximación general o filosófica.

Los métodos son importantes porque introducen una disciplina en el desarrollo de sistemas de software complejos. **Definen los productos que sirven como vehículo común para la comunicación entre miembros de un equipo de desarrollo**.

Diseño estructurado descendente o diseño compuesto: (representado en lenguajes como FORTRAN y COBOL) se aplica descomposición algorítmica para fragmentar un problema grande en pasos pequeños. La unidad fundamental es el subprograma.

Diseño dirigido por los datos: con este diseño, la estructura de un sistema de software se deduce de la correspondencia entre las entradas y las salidas del sistema.

Diseño orientado a objetos: se modelo los sistemas de software como colecciones de objetos que cooperan, tratando los objetos individuales como instancias de una clase que está adentro de una jerarquía de clases. Lenguajes que reflejan esta topología son Smalltalk, Object Pascal, C++, Common Lisp Object System (CLOS) y Ada.

La **descomposición orientada a objetos** produce sistemas más pequeños a través de la **reutilización de mecanismos comunes**, proporcionando una útil **economía de expresión**. Los sistemas orientados a objetos son más **resistentes al cambio** por lo que están **mejor preparados para evolucionar** a través del tiempo, porque está basado en formas intermedias estables (abstracciones y mecanismos ya probados). Está diseñado para evolucionar de forma incremental, partiendo de sistemas más pequeños en los que ya se tiene confianza.

El rol de la abstracción.

Es una técnica muy potente para enfrentarse a la complejidad. Al ser incapaces de dominar en su totalidad a un objeto complejo, **ignoramos entonces sus detalles no esenciales, tratando en su lugar con el modelo generalizado e idealizado del objeto**.

El rol de la jerarquía.

Reconocer jerarquías de clases y objetos es otra forma de incrementar el contenido semántico de bloques de información. la **estructura de objetos** es importante porque nos muestra como los objetos **colaboran entre sí** a través de diferentes mecanismos, mientras que la **estructura de clases resalta la estructura y comportamiento comunes** en el interior del sistema.

La **identificación de jerarquías** en un sistema complejo no es fácil ya que requiere que se descubran patrones entre muchos objetos, cada uno de los cuales puede incluir comportamientos muy complicados.

Puede hacerse frente a las restricciones de la cognición humana mediante el uso de la **descomposición**, la **abstracción** y la **jerarquía**.

1.4 - Del diseño de sistemas complejos.

El significado del diseño.

Es la **aproximación disciplinada que se utiliza para inventar una solución** para algún problema, suministrando así un camino desde los requerimientos hasta la implantación.

Construir un sistema que:

- satisfaga determinada condición funcional
- se ajuste a las limitaciones impuestas por el medio de destino
- respete requisitos implícitos o explícitos sobre el rendimiento y reutilización de recursos
- satisfaga criterios de diseño implícitos o explícitos sobre la forma del artefacto
- satisfaga restricciones sobre el propio proceso de diseño, tales como su longitud o coste, o las herramientas disponibles para realizar el diseño

- "Crear una estructura interna (o arquitectura) clara y relativamente simple"
- "Es un balance entre un conjunto de requisitos que compiten"
- "Los productos del diseño son modelos que permiten razonar sobre las estructuras, hacer concesiones cuando los requisitos entran en conflicto y, en general, proporcionar un anteproyecto para la implementación.

La importancia de construir un modelo.

Cada modelo dentro de un diseño describe un aspecto específico del sistema que se está considerando. Generalmente, se busca construir nuevos modelos basados en modelos viejos en los que ya se confía. Los **modelos dan la oportunidad de fallar en condiciones controladas**, bajo situaciones tanto previstas como improbables para modificarlo para que se comporte del modo esperado o deseado.

Los elementos de los métodos de diseño del software.

1. **Notación** El lenguaje para expresar cada modelo
2. **Proceso** Las actividades que encaminan a la construcción ordenada de los modelos del sistema.
3. **Herramientas** Los artefactos que eliminan el tedio de construir el modelo y reafirman las reglas sobre los propios modelos, de forma que sea posible revelar errores e inconsistencias.

Los modelos del desarrollo orientado a objetos.

El **diseño orientado a objetos** es el método que lleva a una descomposición orientada a objetos. Aplicando diseño orientado a objetos **se crea software resistente al cambio** (mejor evolución) y **escrito con economía de expresión**.

- Modelo lógico
- Modelo físico
- Modelo estático
- Modelo dinámico

RESUMEN – CAPITULO 1

- ✓ El software es complejo de forma innata; la complejidad de los sistemas de software excede frecuentemente la capacidad intelectual humana.
- ✓ La tarea del equipo de desarrollo de software es la de crear una ilusión de sencillez.
- ✓ La complejidad toma a menudo la forma de una jerarquía; esto es útil para modelar tanto las jerarquías “es-un” como “parte de” de un sistema complejo.
- ✓ Los sistemas complejos evolucionan generalmente de formas intermedias estables.
- ✓ Existen factores de limitación fundamentales en la cognición humana; puede hacerse frente a estas restricciones mediante el uso de la descomposición, abstracción y jerarquía.
- ✓ Los sistemas complejos pueden verse centrando la atención bien en las cosas o bien en los procesos; hay razones de peso para aplicar la descomposición orientada a objetos, en la cual se ve el mundo como una colección significativa de objetos que colaboran para conseguir algún comportamiento de nivel superior.
- ✓ El diseño orientado a objetos es el método que conduce a una descomposición orientada a objetos; el diseño orientado a objetos define una notación y un proceso para construir sistemas de software complejos, y ofrece un rico conjunto de modelos lógicos y físicos con los cuales se puede razonar sobre diferentes aspectos del sistema que se esta considerando.

Capítulo 2 – El modelo de objetos

El modelo de objetos es el conjunto de sólidos fundamentos de ingeniería para la tecnología orientada a objetos. Este modelo abarca los principios de **abstracción, encapsulación, modularidad, jerarquía, tipos, concurrencia y persistencia**. Los conceptos no son nuevos pero el modelo los integra de forma sinérgica.

Por desgracia la programación orientada a objetos significa cosas distintas para personas distintas.

2.1 - La evolución del modelo de objetos.

Tendencias en ingeniería de software.

Las generaciones de lenguajes de programación.

- Desplazamiento de programación al por menor a la programación al por mayor.
- La evolución de los lenguajes de alto nivel.

La tendencia ha demostrado un desplazamiento de los lenguajes imperativos hacia los lenguajes declarativos.

Lenguajes imperativos: le dicen a la computadora qué hacer.

Lenguajes declarativos: describen las abstracciones claves del dominio del problema.

Primera generación: representó un paso de acercamiento al problema y un paso de alejamiento de la máquina que había debajo al poder escribir el vocabulario casi totalmente matemático y científico directamente con fórmulas (sin complicaciones de lenguaje de máquina).

Segunda generación: se enfatizó las abstracciones algorítmicas. Lo principal ahora era decirle a la máquina lo que debía hacer.

Tercera generación: se requirieron más tipos de datos. Se empezó a soportar la abstracción de datos.

A partir de los años setenta se crearon más de dos millares de lenguajes de programación de los cuales pocos sobrevivieron.

Topología de los lenguajes de primera y principios de la segunda generación.

1. El bloque principal de construcción de programas de esta época era **el subprograma o párrafo**. Estructura física relativamente plana, consistente solo en datos globales y subprogramas.
2. **Dependencia de cada uno de los subprogramas a los datos globales**. Un error en una parte de estos programas puede tener un devastador efecto de propagación (debido a que las estructuras de datos globales están expuestas a todos los subprogramas).
3. **Difícil mantener la integridad** del diseño original en un sistema grande.
4. La **entropía** es fácil de encontrar. Enorme cantidad de acoplamientos entre subprogramas y flujo de control retorcidos, amenazando la fiabilidad y claridad de la solución del sistema.

Topología de los lenguajes de fines de la segunda y principios de la tercera generación.

Incorporación del concepto de **abstracción** «**procedimental**» con tres consecuencias:

1. mecanismos de **pasaje de parámetros**,
2. se asentaron los **fundamentos de la programación estructurada** y
3. surgieron los **métodos de diseño estructurado** (con subprogramas como bloques físicos básicos para la construcción).

Topología de los lenguajes de finales de la tercera generación.

Se incorporó la idea del módulo compilado separadamente para trabajar en equipos de desarrollo grande y dividir las tareas, pero que carecían de reglas que exigiesen una consistencia semántica entre los diferentes módulos escritos.

Topología de los lenguajes basados en objetos y orientados a objetos.

1. Surgieron primero los **métodos de diseño dirigido por los datos**, que proporcionaron una aproximación disciplinada al hecho de realizar abstracciones de datos en lenguajes orientados algorítmicamente.
2. Aparecieron **teorías acerca del concepto de tipo**.

Estos dos conceptos fueron mejorando poco a poco a través de los millares de lenguajes que perecieron y culminaron representándose en lenguajes como Smalltalk, Object Pascal, C++, CLOS, Ada y Eiffel.

Estos lenguajes son **basados en objetos y orientados a objetos**. El bloque básico de construcción de este tipo de lenguajes es el **módulo** (que representa una colección lógica de clases y objetos en lugar de subprogramas).

Fundamentos del modelo de objetos.

Las **clases** y **objetos** son los bloques básicos de construcción en este modelo.

El modelo de objetos resultó ser un concepto unificador de la informática, al aplicársele a las interfaces de usuario, bases de datos y arquitecturas de computadores. Todo esto debido a los primeros puntos citados del primer capítulo: la **orientación a objetos ayuda a combatir la complejidad inherente a muchos tipos de sistemas**

Un **objeto sólo puede cambiar de estado**, actuar, ser manipulado o permanecer en relación con otros objetos de maneras apropiadas. Existen **propiedades invariantes que caracterizan a un objeto y su comportamiento**.

Programación orientada a objetos.

Es un método de implementación en el que los programas se organizan como colecciones cooperativas de objetos, cada uno de los cuales representa una instancia de alguna clase, y cuyas clases son, todas ellas miembros de una jerarquía de clases unidas mediante relaciones de herencia.

Si falta cualquiera de estos elementos, el programa no es orientado a objetos.

La programación **sin herencia** es explícitamente no orientada a objetos, se denomina **programación con tipos abstractos de datos**.

Un lenguaje es orientado a objetos si y sólo si:

1. soporta objetos que son abstracciones de datos con un interfaz de operaciones con nombre y un estado local oculto
2. los objetos tiene un tipo asociado (clase)
3. los tipos (clase) pueden heredar atributos de los supertipos (superclases).

Para un lenguaje soportar la herencia significa que es posible expresar relaciones «es un» entre tipos.

Basado en objeto es un lenguaje que no ofrece soporte directo para la herencia.

Diseño orientado a objetos.

Es un método de diseño que abarca el proceso de descomposición orientada a objetos y una notación para describir los modelos lógico y físico, así como los modelos estático y dinámico del sistema que se diseña.

El soporte para la descomposición orientada a objetos diferencia a este tipo de diseño del estructurado ya que permite diagramar lógicamente el sistema haciendo uso de abstracciones de clases y objetos.

Análisis orientado a objetos.

Enfatiza la construcción de modelos del mundo real, utilizando una visión orientada a objetos.

Es un método de análisis que examina los requisitos desde la perspectiva de las clases y objetos que se encuentran en el vocabulario del dominio del problema.

Los productos del **AOO** sirven como modelos de los que se puede partir para un DOO.

Los productos del **DOO** puede utilizarse como anteproyectos para la implementación completa de un sistema utilizando métodos de **POO**.

2.2 - Elementos del modelo de objetos.

Tipos de paradigmas de programación.

Estilo de programación: una forma de organizar programas sobre las bases de algún modelo conceptual de programación y un lenguaje apropiado para que resulten claros los programas escritos en ese estilo.

Estilo de programación	Abstracción que se usa
Orientado a procedimientos	Algoritmos
Orientado a objetos	Clases y objetos
Orientado a lógica	Objetivos, expresados como cálculo de predicados
Orientado a reglas	Reglas si-entonces
Orientados a restricciones	Relaciones invariantes

Hay cuatro elementos fundamentales del modelo de objetos (si alguno de ellos falta el modelo no es orientado a objetos):

- **Abstracción**
- **Encapsulamiento**
- **Modularidad**
- **Jerarquía**

Hay tres elementos secundarios:

- **Tipos (tipificación)**
- **Concurrencia**
- **Persistencia**

Abstracción.

Surge del reconocimiento de las similitudes entre ciertos objetos, situaciones o procesos del mundo real, y la decisión de concentrarse en esas similitudes e ignorar por el momento las diferencias.

Una abstracción denota las características esenciales de un objeto que lo distinguen de todos los demás tipos de objeto y proporciona así fronteras conceptuales nítidamente definidas respecto a la perspectiva del observador.

Una abstracción se centra en la visión externa del objeto y por tanto sirve para separar el comportamiento esencial de un objeto de su implantación (= **barrera de abstracción**).

Abstracción de entidades	Un objeto representa un modelo útil de una entidad del dominio del problema o del dominio de la solución.
Abstracción de acciones	Un objeto que proporciona un conjunto generalizado de operaciones y todas ellas desempeñan funciones del mismo tipo.
Abstracción de máquinas virtuales	Un objeto que agrupa operaciones, todas ellas virtuales utilizadas por algún nivel superior o control, u operaciones que utilizan todas algún conjunto de operaciones de nivel inferior.
Abstracción de coincidencia	Un objeto que almacena un conjunto de operaciones que no tiene relación entre sí.

Un **cliente** es cualquier objeto que utiliza los recursos de otro objeto (denominado **servidor**).

Se puede caracterizar el comportamiento de un objeto considerando los servicios que presta a otros objetos.

Al conjunto completo de operaciones que puede realizar un cliente sobre un objeto, junto con las formas de invocación u órdenes que admite, se le denomina **protocolo**.

Un **invariante** es una condición booleana cuyo valor de verdad debe mantenerse.

Precondiciones: invariantes asumidos por la operación

Postcondiciones: invariantes satisfechos por la operación.

Si se viola una precondición significa que un cliente no ha satisfecho su parte del convenio, y por tanto el servidor no puede proceder con fiabilidad.

Si se viola una postcondición significa que un servidor no ha llevado a cabo su parte del contrato, y por tanto sus clientes ya no pueden seguir confiando en el comportamiento del servidor.

Una «**excepción**» es una indicación de que **no** se ha satisfecho algún invariante.

Todas las abstracciones tiene propiedades tanto **estáticas** como **dinámicas**.

Objeto: archivo en un disco.

Propiedades: **ESTÁTICAS** → nombre, tamaño, contenido.

Valores de las propiedades: **DINÁMICOS** → su nombre, su tamaño y su contenido pueden cambiar.

Encapsulamiento.

El significado del encapsulamiento.

Ninguna parte de un sistema complejo debe depender de los detalles internos de otras partes.

Mientras la abstracción ayuda a las personas a pensar sobre lo que están haciendo, el encapsulamiento permite que los cambios hechos en los programas sean fiables con el menor esfuerzo.

El encapsulamiento se consigue a menudo mediante la **ocultación de información**, la estructura de un objeto está oculta así como la implantación de sus métodos.

«Para que la abstracción funcione, la implementación debe estar encapsulada»

En la práctica cada clase debe tener dos partes: un interfaz y una implementación.

Interfaz: captura sólo su **vista externa**, abarcando la abstracción que se ha hecho del comportamiento común de todas las instancias de esa clase.

Implementación: (o implantación) comprende la **representación de la abstracción así como los mecanismos** que consiguen el comportamiento deseado.

El encapsulamiento es el proceso de almacenar en un mismo compartimento los elementos de una abstracción que constituyen su estructura y su comportamiento; sirve para separar el interfaz contractual de una abstracción y su implementación.

Esos elementos encapsulados se llaman «**secretos de una abstracción**».

La ocultación es un concepto **relativo**: lo que está oculto a un nivel de abstracción puede representar la visión externa a otro nivel de abstracción.

Modularidad.

El significado de la modularidad.

«El acto de fragmentar un programa en componentes individuales puede reducir su complejidad en algún grado». Esto es una razón útil, pero más poderosa es la justificación de que la fragmentación **crea una serie de fronteras bien definidas y documentadas dentro del programa**. El uso de módulos es esencial para ayudar a manejar la complejidad.

La modularización consiste en dividir un programa en módulos que pueden compilarse separadamente, pero que tienen conexiones con otros módulos. Las conexiones entre módulos son las suposiciones que cada módulo hace acerca de todos los demás.

La decisión sobre el conjunto adecuado de módulos para determinado problema es casi tan difícil como decidir sobre el conjunto adecuado de abstracciones.

En el diseño estructurado tradicional, la modularización persigue ante todo el agrupamiento significativo de subprogramas, utilizando los criterios de acoplamiento y cohesión. En el diseño orientado a objetos, el

problema presenta diferencias sutiles, la tarea es decidir **dónde empaquetar físicamente las clases y objetos** a partir de la estructura lógica del diseño, y éstos son claramente diferentes de los subprogramas.

El objetivo de fondo de la descomposición en módulos es la reducción del coste del software al permitir que los módulos se diseñen y revisen independientemente. La estructura de cada módulo debería ser lo bastante simple como para ser comprendida en su totalidad.

Los detalles de un sistema, que probablemente cambien de forma independiente, deberían ser secretos en módulos separados; las únicas suposiciones que deberían darse entre módulos son aquellas cuyo cambio se considera improbable.

Hay que hacer lo posible por construir **módulos cohesivos** (agrupando abstracciones que guarden cierta relación lógica) y **débilmente acoplados** (minimizando las dependencias entre módulos).

La modularidad es la propiedad que tiene un sistema que ha sido descompuesto en un conjunto de módulos cohesivos y débilmente acoplados.

Jerarquía.

El significado de la jerarquía.

La **abstracción** es algo bueno, pero excepto en las aplicaciones más triviales, puede haber muchas más abstracciones diferentes de las que se pueden comprender simultáneamente.

El **encapsulamiento** ayuda a manejar esta complejidad ocultando la visión interna de las abstracciones.

La **modularidad** también ayuda, ofreciendo una vía para agrupar abstracciones relacionadas lógicamente.

La jerarquía es una ordenación o clasificación de abstracciones.

Las dos jerarquías más importantes en un sistema complejo son su estructura de clases (la jerarquía «**de clases**») y su estructura de objetos (la jerarquía «**de partes**»).

La **herencia** es la jerarquía de clases más importante (elemento esencial en los sistemas orientados a objetos). Básicamente la herencia define una relación entre clases, en la que una clase comparte la estructura de comportamiento definida en una o más clases (lo denominado **herencia simple** o **múltiple**).

Una subclase hereda de una o más superclases. Típicamente una subclase aumenta o redefine la estructura y el comportamiento de sus superclases. Semánticamente la herencia denota una relación de tipo «**es un**».

A medida que se desarrolla una jerarquía de herencias, la estructura y comportamiento comunes a diferentes clases tenderá a migrar hacia superclases comunes.

Herencia = jerarquía de generalización / especialización.

- Las **superclases** representan abstracciones generalizadas
- Las **subclases** representan especializaciones en la que los campos y métodos de las superclases sufren adiciones, modificaciones o incluso ocultaciones.

De este modo la herencia permite declarar las abstracciones con economía de expresión.

Sin herencia cada clase sería una unidad independiente, desarrollada partiendo de cero.

La herencia posibilita la **definición de nuevo software** de la misma forma que se presenta un nuevo concepto a un recién llegado, comparándolo con lago que ya le resulte familiar.

«La abstracción de datos intenta proporcionar una barrera opaca tras de la cual se ocultan los métodos y el estado; la herencia requiere abrir ese interfaz en cierto grado y puede permitir el acceso a métodos y al estado sin abstracción»

Para la **herencia múltiple** primero se idean clases que capturen independientemente las propiedades buscadas (se las llama clases aditivas) y luego se las une en una subclase. Hay problemas de colisión de nombres y herencia repetida.

Jerarquía de agregación: son relaciones «**parte de**». Esto permite hablar de **niveles de abstracción**. Niveles más altos o más bajos para indicar la dependencia de una clase a otra.

Una clase está a nivel más alto de abstracción que cualquiera de las clases que constituyen su implantación.

La combinación de herencia con agregación es muy potente. La agregación permite el agrupamiento físico de estructuras relacionadas lógicamente, y la herencia permite que estos grupos de aparición frecuente se reutilicen con facilidad en diferentes abstracciones.

Tipos. Tipificación.

El significado de los tipos.

Un tipo es una caracterización precisa de las propiedades estructurales o de comportamiento que comparten una serie de entidades.

(Las clases implementan a los tipos) No son exactamente lo mismo.

Los tipos son la puesta en vigor de la clase de los objetos, de modo que los objetos de tipos distintos no pueden intercambiarse o, como mucho, pueden intercambiarse sólo de formas muy restringidas.

Un lenguaje de programación determinado puede tener comprobación estricta de tipos, comprobación débil de tipos, o incluso no tener tipos, y aún así ser considerado como orientado a objetos.

Comprobación estricta de tipos: no puede llamarse a una operación sobre un objeto a menos que en la clase o superclases del objeto esté definido el prototipo de esa operación. Todas las expresiones son consistentes respecto al tipo.

En lenguajes con comprobación estricta de tipos puede detectarse en tiempo de compilación cualquier violación a la concordancia de tipos.

Comprobación débil de tipos: hacer cumplir la clase de un objeto, lo que previene el intercambio de objetos de diferentes tipos o, como mucho, permite este intercambio de maneras muy limitadas.

Beneficios del uso de tipos estrictos:

- sin la comprobación de tipos un programa puede estallar de forma misteriosa en ejecución en la mayoría de los lenguajes.
- en la mayoría de los sistemas, el ciclo editar-compilar-depurar es tan tedioso que la comprobación temprana de errores es indispensable.
- la declaración de tipos ayuda a documentar los programas.
- la mayoría de los compiladores pueden generar un código más eficiente si se han declarado tipos.

Ligadura estática y dinámica.

a. **Ligadura estática o temprana**: significa que se fijan los tipos de todas las variables y expresiones en tiempo de compilación;
b. **Ligadura dinámica o tardía**: los tipos de las variables y expresiones no se conocen hasta el tiempo de ejecución.

Polimorfismo: representa un concepto de teoría de tipos, en el que un solo nombre (tal como una declaración de variable) puede denotar objetos de muchas clases diferentes que se relacionan por alguna superclase común de operaciones. El opuesto del polimorfismo es el **monomorfismo**.

Concurrencia.

El significado de la concurrencia.

Hilo de control: un solo proceso.

Es común tener que manejar muchos eventos diferentes simultáneamente. Un solo proceso es la raíz a partir de la cual se producen acciones dinámicas independientes dentro del sistema.

Los sistemas que se ejecutan en múltiples CPUs permiten hilos de control verdaderamente concurrentes, mientras que los sistemas que se ejecutan en una sola CPU sólo pueden conseguir la ilusión de hilos concurrentes de control, normalmente mediante algún algoritmo de tiempo compartido.

Proceso pesado: aquel típicamente manejado de forma independiente por el sistema operativo de destino, y abarca su propio espacio de direcciones.

Proceso ligero: suele existir dentro de un solo proceso del sistema operativo en compañía de otros procesos ligeros, que comparten el mismo espacio de direcciones.

Diseñar un sistema que abarque múltiples hilos de control es muy difícil ya que hay que preocuparse de problemas tales como interbloqueo, bloqueo activo, inanición, exclusión mutua y condiciones de competencia.

Mientras que la POO se centra en la abstracción de datos, encapsulamiento y herencia, la concurrencia se centra en la abstracción de procesos y la sincronización. El objeto es un concepto que unifica estos dos puntos de vista distintos: cada objeto (extraído de una abstracción del mundo real) puede representar un hilo separado de control (una abstracción de un proceso). Tales objetos se llaman **activos**.

En un sistema OO se puede conceptualizar el mundo como un conjunto de objetos cooperativos, algunos de los cuales son activos y sirven así como centros de la actividad independiente.

La concurrencia es la propiedad que distingue un objeto activo de uno que no está activo.

Una vez que se introduce la concurrencia a un sistema, hay que tener en cuenta como los objetos activos sincronizan sus actividades con otros, así como con objetos puramente secuenciales.

Persistencia.

El significado de la persistencia.

Un objeto de software ocupa una cierta cantidad de espacio, y existe durante una cierta cantidad de tiempo. Hay un espacio de existencia que va desde los objetos transitorios que surgen en la evaluación de una expresión hasta los objetos de una base de datos. Este espectro de persistencia abarca lo siguiente:

1. Resultados transitorios en la **evaluación de expresiones**.
2. **Variables locales** en la activación de procedimientos.
3. **Variables propias**, **globales** y **elementos del montículo** cuya duración difiere de su ámbito.
4. Datos que existen **entre ejecuciones de un programa**.
5. Datos que existen **entre varias versiones de un programa**.
6. Datos que **sobreviven al programa**.

La persistencia abarca algo más que la mera duración de los datos. En las bases de datos orientadas a objetos, no sólo persiste el estado del objeto, sino que su clase debe trascender también a cualquier programa individual, de forma que todos los programas interpreten de la misma manera el estado almacenado. Esto hace que sea un reto evidente el mantener la integridad de una base de datos a medida que crece, particularmente si hay que cambiar la clase de un objeto.

La persistencia es la propiedad de un objeto por la que su existencia trasciende el tiempo 8es decir, el objeto continúa existiendo después de que su creador deja de existir) y / o el espacio (es decir, la posición del objeto varía con respecto al espacio de direcciones en el que fue creado).

2.3 - Aplicación del modelo de objetos.

Beneficios del modelo de objetos.

El uso del modelo de objetos conduce a la construcción de sistemas que incluyen los cinco atributos de los sistemas complejos bien estructurados.

1) Ayuda además a explotar la potencia expresiva de los lenguajes de programación basados en objetos y orientados a objetos. Se han logrado mejoras significativas con por ejemplo, una pizca de abstracción de datos añadida donde era claramente útil, o apreciando las jerarquías de clases en el proceso de diseño.
2) Promueve la reutilización no sólo de software, sino de diseños enteros, conduciendo a la creación de marcos de desarrollo de aplicaciones reutilizables. Los sistemas orientados a objetos son frecuentemente más pequeños que sus implantaciones equivalente no orientadas a objetos. Significa escribir y mantener menos código aparte de una reutilización de software que refleja beneficios de coste y planificación.
3) Produce sistemas que se construyen sobre formas intermedias estables, son más flexibles al cambio. Tienen una mejor evolución en el tiempo y mayor ciclo de vida.
4) Reduce los riesgos inherentes al desarrollo de sistemas complejos. La integración se distribuye a lo largo del ciclo vital en vez de suceder como un evento principal.
5) Resulta atractivo para el funcionamiento de la cognición humana, porque muchas personas que no tiene ni idea de cómo funciona un computador encuentran bastante natural la idea de los sistemas orientados a objetos.

El modelo de objetos ha demostrado ser aplicable a una amplia variedad de dominios de problema. Hoy en día, el DOO puede que sea el único método que puede emplearse para atacar la complejidad innata a muchos sistemas grandes. Sin embargo, puede no ser aconsejable en dominios, no por razones técnicas sino por cuestiones como falta de personal con entrenamiento adecuado o buenos entornos de desarrollo.

RESUMEN – CAPITULO 2

- ✓ La madurez de la ingeniería del software ha conducido al desarrollo de métodos de análisis, diseño y programación orientados a objetos, todos los cuales tienen la misión de resolver los problemas de la programación a gran escala.
- ✓ Existen varios paradigmas de programación distintos: orientados a procedimientos, orientados a objetos, orientados a lógica, orientados a reglas y orientados a restricciones.
- ✓ Una abstracción denota las características esenciales de un objeto que lo distinguen de todos los demás tipos de objeto, y proporciona así fronteras conceptuales definidas con nitidez, desde la perspectiva del observador.
- ✓ El encapsulamiento es el proceso de compartimentar los elementos de una abstracción que constituyen su estructura y comportamiento. Sirve para separa el interfaz “contractual” de una abstracción y su implantación.
- ✓ La modularidad es la propiedad de un sistema que ha sido descompuesto en un conjunto de módulos cohesivos y débilmente acoplados.
- ✓ La jerarquía es una graduación u ordenación de abstracciones.
- ✓ Los tipos son el resultado de imponer la clase de los objetos, de forma que los objetos de tipos diferentes no pueden intercambiarse o, como mucho, pueden hacerlo de formas muy restrictivas.
- ✓ La concurrencia es la propiedad que distingue un objeto activo de uno que no está activo.
- ✓ La persistencia es la propiedad de un objeto mediante la cual su existencia perdura en el tiempo y/o el espacio.

Capítulo 3 – Clases y Objetos

Las clases y los objetos son los bloques básicos de construcción de un sistema de software complejo.

3.1 - La naturaleza de los objetos.

Qué es y qué no es un objeto.

A través del concepto de objeto un niño llega a darse cuenta de que los objetos tienen una permanencia e identidad además de cualesquiera operaciones sobre ellos.

Desde la perspectiva de la cognición humana un objeto es cualesquiera de las siguientes cosas:

- Una cosa tangible y / o visible.
- Algo que puede comprenderse intelectualmente.
- Algo hacia lo que se dirige un pensamiento o acción.

Un objeto modelo alguna parte de la realidad, y es por tanto algo que existe en el tiempo y el espacio.

Los objetos del mundo real no son el único tipo de objeto de interés para el desarrollo del software. Otros tipos importantes de objetos son invenciones del proceso de diseño cuyas colaboraciones con otros objetos semejantes sirven como mecanismos para desempeñar algún comportamiento de nivel superior.

Un objeto representa un elemento, unidad o entidad individual e identificable, ya sea real o abstracta, con un papel bien definido en el dominio del problema.

Un objeto es cualquier cosa que tenga una frontera definida con nitidez.

Algunos objetos pueden tener **límites conceptuales precisos**, pero aún así pueden **representar eventos o procesos intangibles**.

Algunos objetos pueden ser **tangibles** y aún así tener **fronteras físicas difusas** (objetos como los ríos, la niebla, etc.).

*Un **objeto** tiene estado, comportamiento e identidad; la estructura y comportamiento de objetos similares están definidos en su clase común; los términos **instancia** y **objeto** son **intercambiables**.*

Estado.

Semántica.

El comportamiento de un objeto está influenciado por su historia: el orden en el que se opera sobre un objeto es importante. La **razón para este comportamiento dependiente del tiempo y los eventos es la existencia de un estado interior del objeto**. O sea que:

El estado de un objeto abarca todas las propiedades (normalmente estáticas) del mismo más los valores actuales (normalmente dinámicos) de cada una de esas propiedades.

Una propiedad es una característica distintiva o inherente, un rasgo o cualidad que contribuye a hacer que un objeto sea ese objeto y no otro. Todas las propiedades tiene algún valor. Este valor puede ser una mera cantidad, o puede denotar a otro objeto. Esas **cantidades** son intemporales, inmutables y no instanciadas; mientras que los **objetos** existen en el tiempo, son modificables, tienen estado, son instanciados y pueden crearse, destruirse y compartirse.

El hecho de que todo objeto tiene un estado implica que todo objeto toma cierta cantidad de espacio, ya sea en el mundo físico o en la memoria del computador.

Análisis y Diseño Orientado a Objetos – Grady Booch

Puede decirse que **todos los objetos de un sistema encapsulan algún estado**, y que **todo el estado de un sistema está encapsulado en objetos**. Sin embargo, encapsular el estado de un objeto es un punto de partido pero no es suficiente para permitir que se capturen todos los designios de las abstracciones que se descubren e inventan durante el desarrollo. Por esta razón, **hay que considerar también cómo se comportan los objetos**.

Comportamiento.

El significado del comportamiento.

Ningún objeto existe de forma aislada. Los objetos reciben acciones y actúan sobre otros objetos.

El comportamiento es cómo actúa y reacciona un objeto, en términos de sus cambios de estado y paso de mensajes. Representa su actividad visible y comprobable exteriormente.

Una operación es una acción que un objeto efectúa sobre otro con el fin de provocar una reacción. Los términos **operación** y **mensaje** son intercambiables.

En la mayoría de los lenguajes de programación orientados a objetos, las operaciones que los clientes pueden realizar sobre un objeto suelen declararse como **métodos** (o función miembro en C++ por ejemplo).

La definición de **comportamiento** también recoge que el estado de un objeto afecta asimismo a su comportamiento. El comportamiento de un objeto es función de su estado así como también de la operación que se realiza sobre él, teniendo algunas operaciones el efecto lateral de modificar el estado del objeto.

El estado de un objeto representa los resultados acumulados de su comportamiento.

Una operación denota un servicio que una clase ofrece a sus clientes. Los tres tipos más comunes de operaciones sobre un objeto son:

a. **Modificador** Una operación que altera el estado de un objeto.
b. **Selector** Una operación que accede al estado de un objeto, pero no altera ese estado.
c. **Iterador** Una operación que permite acceder a todas las partes de un objeto en algún orden perfectamente establecido.
d. **Constructor** Una operación que crea un objeto y / o inicializa su estado.
e. **Destructor** Una operación que libera el estado de un objeto y / o destruye el propio objeto.

Roles y responsabilidades.

Todos los métodos y subprogramas libres asociados a un objeto concreto forman su protocolo. El protocolo define así la envoltura del comportamiento admisible en un objeto, englobando la visión estática y dinámica completa del mismo.

Para la mayoría de las abstracciones es útil dividir este protocolo en grupos lógicos de comportamiento. Estas colecciones, que constituyen una partición del espacio de comportamiento de un objeto, denotan papeles que un objeto puede desempeñar. Un papel es una máscara que se pone un objeto y por tanto define un contrato entre una abstracción y sus clientes.

Las responsabilidades de un objeto «incluyen dos elementos clave»: el conocimiento que un objeto mantiene y las acciones que puede llevar a cabo. **Las responsabilidades están encaminadas a transmitir un sentido del propósito de un objeto y de su lugar en el sistema**. Las responsabilidades de un objeto son todos los servicios que proporciona para todos los contratos que soporta.

Los objetos como máquinas.

La existencia de un estado significa que el orden en el que se invocan las operaciones es importante. Esto da lugar a la idea de que cada objeto es como una pequeña máquina independiente. Se pueden entonces

clasificar los objetos como activos o pasivos. Un **objeto activo** es aquel que comprende su propio hilo de control, mientras que un **objeto pasivo** no.

Los objetos activos suelen ser autónomos, pueden exhibir algún comportamiento sin que ningún otro objeto opere sobre ellos. Los pasivos, sólo pueden padecer un cambio de estado cuando se actúa explícitamente sobre ellos.

Identidad.

Semántica.

La identidad es aquella propiedad de un objeto que lo distingue de todos los demás objetos.

La mayoría de los lenguajes de programación y bases de datos utilizan nombres de variable para distinguir objetos temporales, mezclando la posibilidad de acceder a ellos con su identidad. El fracaso en reconocer la diferencia entre el nombre del objeto y el objeto en sí mismo es fuente de muchos errores en la programación orientada a objetos.

Copia, asignación e igualdad.

La compartición estructural se da cuando la identidad de un objeto recibe un alias a través de un segundo nombre. Es una operación de **copia**.

La **asignación** es del mismo modo en general una operación de copia.

El problema de la **igualdad** está en relación muy estrecha con el de la asignación. Aunque se presenta como un concepto simple la igualdad puede presentar una de dos cosas.

Primero, puede significar que dos nombres designan el mismo objeto y segundo, puede significar que dos nombres designan objetos distintos cuyos estados son iguales.

Espacio de vida de un objeto.

El tiempo de vida de un objeto se extiende desde el momento en que se crea por primera vez (y consume así espacio por primera vez) hasta que ese espacio se recupera. Para crear explícitamente un objeto hay que declararlo o bien asignarle memoria dinámicamente.

3.2 - Relaciones entre objetos.

Tipos de relaciones.

Los objetos contribuyen al comportamiento de un sistema colaborando con otros. En lugar de un procesador triturador de bits que golpea y saquea estructuras de datos, tenemos un universo de objetos bien educados que cortésmente solicitan a los demás que lleven a cabo sus diversos deseos. La relación entre dos objetos cualesquiera abarca las suposiciones que cada uno realiza acerca del otro, incluyendo qué operaciones pueden realizarse y qué comportamiento se obtiene. Hay dos tipos de jerarquías de objetos:

- Enlaces
- Agregación

Enlaces.

Una conexión física o conceptual entre objetos. Un objeto colabora con otros objetos a través de sus enlaces con éstos. Un enlace denota la asociación específica por la cual un objeto (el cliente) utiliza los servicios de otro objeto (el suministrador o servidor), o a través de la cual un objeto puede comunicarse con otro.

El paso de mensajes entre dos objetos es unidireccional, aunque ocasionalmente puede ser bidireccional.

Como participante de un enlace, un objeto puede desempeñar uno de tres papeles:

1. **Actor** Un objeto que puede operar sobre otros objetos pero nunca se opera sobre él por parte de otros objetos; en algunos contextos, los términos objeto activo y actor son equivalentes.
2. **Servidor** Un objeto que nunca opera sobre otros objetos; sólo otros objetos operan sobre él.
3. **Agente** Un objeto que puede operar sobre otros objetos y además otros objetos pueden operar sobre él; un agente se crea normalmente para realizar algún trabajo en nombre de un actor u otro agente.

Visibilidad.

Considérense dos objetos A y B con un enlace entre ambos. Para que A pueda enviarle un mensaje a B, B debe ser visible para A de algún modo. Durante el análisis del problema se puede ignorar la visibilidad, pero una vez que se comienza a idear implantaciones concretas hay que considerarla a través de los enlaces, porque las decisiones en este punto dictan el ámbito y acceso de los objetos a cada lado del enlace.

Hay cuatro formas en que un objeto puede tener visibilidad para otro:

- **El objeto servidor es global para el cliente.**
- **El objeto servidor es un parámetro de alguna operación del cliente.**
- **El objeto servidor es parte del objeto cliente.**
- **El objeto servidor es un objeto declarado localmente en alguna operación del cliente.**

Sincronización.

Siempre que un objeto pasa un mensaje a otro a través de un enlace se dice que los dos objetos están **sincronizados**. Para objetos de una aplicación completamente secuencial, esta sincronización suele realizarse mediante una simple invocación de métodos. Sin embargo en presencia de múltiples hilos de control, los objetos requieren un paso de mensajes más sofisticado. Hay que elegir uno de tres enfoques:

1. **Secuencial** La semántica del objeto pasivo está garantizada sólo en presencia de un único objeto activo simultáneamente.
2. **Vigilado** La semántica del objeto pasivo está garantizada en presencia de múltiples hilos de control, pero los clientes activos deben colaborar para lograr la exclusión mutua.
3. **Síncrono** La semántica del objeto pasivo está garantizada en presencia de múltiples hilos de control, y el servidor garantiza la exclusión mutua.

Agregación.

Mientras que los enlaces denotan relaciones de igual-a-igual o cliente / servidor, la agregación denota una jerarquía todo / parte, con la capacidad de ir desde el todo (también llamado el agregado) hasta sus partes (conocido también como atributos).

La agregación puede o no denotar contención física.

Existen claros pros y contras entre los enlaces y la agregación. La agregación es a veces mejor porque encapsula partes y secretos del todo. A veces son mejores los enlaces porque permiten acoplamientos más débiles entre los objetos.

Por implicación, un objeto que es atributo de otro tiene un enlace a su agregado. A través de este enlace, el agregado puede enviar mensajes a sus partes.

3.3 - La naturaleza de una clase.

Qué es y qué no es una clase.

Los conceptos de clase y objeto están estrechamente ligados pero hay que saber que mientras un objeto es una entidad concreta que existe en el tiempo y el espacio, una clase representa sólo una abstracción, la «esencia» de un objeto (lo que representa las características comunes de todos los objetos que pertenecen a una clase).

Una clase es un conjunto de objetos que comparten una estructura común y un comportamiento común.

Un solo objeto no es más que una instancia de una clase. Un objeto no es una clase, aunque una clase sí puede ser un objeto. La clase es un vehículo necesario, pero no suficiente, para la descomposición.

Interfaz e implementación.

Un objeto es una entidad concreta que desempeña algún papel en el sistema global y una clase captura la estructura y comportamiento comunes a todos los objetos relacionados. Así, una clase sirve como una especie de contrato que vincula a una abstracción y todos sus clientes. Esta visión de la programación como un contrato lleva a distinguir entre la visión externa y la visión interna de una clase. El **interfaz** de una clase proporciona su visión externa y por tanto enfatiza la abstracción a la vez que oculta su estructura y los secretos de comportamiento. Se compone principalmente de todas las operaciones aplicables a instancias de esta clase, pero también puede incluir la declaración de otras clases, constantes, variables, expresiones, según se necesiten para completar la abstracción.

Por contraste, la **implementación** es su visión interna que engloba los secretos de su comportamiento. La implementación de una clase se compone principalmente de la implementación de todas las operaciones definidas en el interfaz de la misma. El interfaz puede dividirse en:

1. **Público** (Public) Una declaración accesible a todos los clientes.
2. **Protegida** (Protected) Una declaración accesible sólo a la propia clase, sus subclases, y sus clases amigas.
3. **Privada** (Private) Una declaración accesible sólo a la propia clase y sus clases amigas.

Ciclo de vida de las clases.

Se puede llegar a la comprensión del comportamiento de una clase simple con sólo comprender la semántica de sus distintas operaciones públicas de forma aislada. Sin embargo, el comportamiento de clases más interesantes implica la interacción de sus diversas operaciones a lo largo del tiempo de vida de cada una de sus instancias.

3.4 - Relaciones entre clases.

Tipos de relaciones.

Las clases, al igual que los objetos no existen aisladamente. Para un domino de problema específico las abstracciones clave suelen estar relacionadas por vías muy diversas e interesantes formando la estructura de clases del diseño.

Se establecen relaciones entre clases porque

a. la relación podría indicar algún tipo de compartición.
b. la relación podría indicar algún tipo de conexión semántica.

Existen tres tipos básicos de relaciones entre clases:

1. **generalización / especialización**; denota una relación «es un».
2. **todo / parte**; denota una relación «parte de».
3. **asociación**; denota alguna dependencia semántica entre clases de otro modo independientes.

Relaciones:

1) Asociación
2) Herencia
3) Agregación
4) Uso
5) Instanciación (creación de instancias o ejemplares)
6) Metaclase

Asociación.

Una asociación sólo denota una dependencia semántica y no establece la dirección de esta dependencia (a menos que se diga lo contrario, una asociación implica relación bidireccional) ni establece la forma exacta en que una clase se relaciona con otra (sólo puede denotarse esta semántica nombrando el papel que desempeña cada clase en relación con la otra).

Cardinalidad.

Existen tres tipos de Cardinalidad en una asociación:

1. Uno a uno
2. Uno a muchos
3. Muchos a muchos.

Herencia.

La herencia es una relación entre clases en la que una clase comparte la estructura y / o comportamiento definidos en una (**herencia simple**) o más clases (**herencia múltiple**). La clase de las que otras heredan se llama **superclase**. La clase que hereda de otra o más clases se denomina subclase. La herencia define por tanto una jerarquía «de tipos» entre clases, en las que una subclase hereda de una o más superclases. Dadas las clases A y B, si A no «es un» tipo de B, entonces A no debería ser una subclase de B.

La capacidad de un lenguaje para soportar o no este tipo de herencia distingue a los lenguajes de programación orientados a objetos de los lenguajes basados en objetos.

Una subclase generalmente aumenta o restringe la estructura y comportamiento existentes en sus superclases. Una subclase que aumenta sus superclases se dice que utiliza **herencia por extensión**, mientras que la subclase que restringe a sus superclases se dice que utiliza **herencia por restricción**.

Se espera que algunas de las clases generadas tengan instancias y que otras no las tengan (frecuentemente las intermedias).

Las clases sin instancias se llaman **clases abstractas**.

Las clases más especializadas se llaman **clases hoja** o **clases concretas**.

La clase más generalizada en una estructura de clases se llama **clase base**.

Una clase cualquiera tiene típicamente dos tipos de clientes:

A. Instancias.

B. Subclases.

Para comprender el significado de una clase particular muchas veces hay que estudiar todas sus superclases, a veces incluyendo sus vistas internas.

Polimorfismo simple.

Es un concepto de teoría de tipos en el que un nombre puede denotar instancias de muchas clases diferentes en tanto y en cuanto estén relacionadas por alguna superclase común. Cualquier objeto denotado por este nombre es, por tanto, capaz de responder a algún conjunto de operaciones de diversas formas.

Sobrecarga es una forma de llamar a un polimorfismo por el cual símbolos como el «+» podrían definirse para significar cosas distintas.

El polimorfismo es más útil cuando existen muchas clases con los mismos protocolos.

Herencia múltiple.

Con herencia simple, cada subclase tiene exactamente una superclase. Esto limita la aplicabilidad de las clases predefinidas, haciendo muchas veces necesario duplicar código.

Las colisiones de nombres pueden aparecer cuando dos o más superclases diferentes utilizan el mismo nombre para algún elemento de sus interfaces, como las variables de instancias o los métodos.

Otro problema es la herencia repetida, que es lo que sucede cuando una clase es un antecesor de otra por más de una vía.

La existencia de herencia múltiple plantea un estilo de **clases**, llamadas **aditivas** porque se mezclan, se combinan pequeñas clases para construir clases con un comportamiento más sofisticado. Una clase aditiva es sintácticamente idéntica una normal, pero su intención es distinta. El propósito de tal clase es únicamente añadir funciones a otras clases. No pueden existir por sí mismas, se usan para aumentar el significado de otra clase. Una clase que se construye principalmente heredando de clases aditivas y no añade su propia estructura o comportamiento se llama **clase agregada**.

Agregación.

Las relaciones de agregación entre clases tienen un paralelismo directo con las relaciones de agregación entre los objetos correspondientes a esas clases.

Contención física.

Contención por valor: un tipo de contención física que significa que el objeto no existe independientemente de la instancia que lo encierra. El tiempo de vida de ambos objetos está en íntima conexión.

Contención por referencia: tipo de agregación menos directo. En este caso, la clase sigue denotando al todo, y una de sus partes sigue siendo una instancia de la clase, aunque ahora hay que acceder a esa parte indirectamente. Sus tiempos de vida no están tan estrechamente emparejados.

La agregación establece una dirección en la relación todo / parte. La contención por valor no puede ser **cíclica** (ambos objetos no pueden ser físicamente partes del otro), aunque la contención por referencia sí puede serlo.

La herencia múltiple se confunde a menudo con la agregación. Cuando se considera la herencia versus agregación, recuérdese aplicar la prueba correspondiente para ambas. Si no se puede afirmar sinceramente que existe una relación «es un» entre dos clases, habría que utilizar agregación o alguna otra relación en vez de herencia.

Uso.

Una clase usa los servicios de otra.

Mientras que una asociación denota una conexión semántica bidireccional, una relación «**de uso**» es un posible **refinamiento de una asociación**, por el que se establece qué abstracción es el cliente y qué abstracción es el servidor que proporciona ciertos servicios.

Instanciación.

Creación de instancias.

Metaclases.

Es una clase cuyas instancias son, ellas mismas, clases. Se justifica su necesidad al hacer notar que en un sistema bajo desarrollo, una clase proporciona al programador un interfaz para comunicarse con la definición de objetos. Para este uso de las clases, es extremadamente útil para ellas ser objetos, de forma que puedan ser manipulados de la misma forma que todas las demás descripciones.

3.5 - La interacción entre clases y objetos.

Relaciones entre clases y objetos.

Las clases y objetos son conceptos diferentes pero en íntima relación. Concretamente todo objeto es instancia de alguna clase y toda clase tiene cero o más instancias.

El papel de clases y objetos en análisis y diseño.

Durante el análisis y las primeras etapas de diseño, el desarrollador tiene dos tareas principales:

- Identificar las clases y objetos que forman el vocabulario del dominio del problema.
- Idear las estructuras por las que conjuntos de objetos trabajan juntos para lograr los comportamientos que satisfacen los requerimientos del problema.

En conjunto se llama a esas clases y objetos las **abstracciones claves** del problema.

Se denomina a esas estructuras cooperativas los **mecanismos** de la implantación.

Durante estas fases del desarrollo, el interés principal del desarrollo de estar en la vista externa de estas abstracciones clave y mecanismos. Esta vista representa el marco de referencia lógico del sistema y, por tanto, abarca la estructura de clases y la estructura de objetos del mismo. En las etapas finales del diseño y entrando ya en la implantación, la tarea del desarrollador cambia: el centro de atención está en la vista interna de estas abstracciones clave y mecanismos, involucrando a su representación física. Pueden expresarse estas decisiones de diseño como parte de la arquitectura de módulos y la arquitectura de procesos del sistema.

3.6 - De la construcción de clases y objetos de calidad.

Medida de la calidad de una abstracción.

¿Cómo puede saberse si una clase u objeto está bien diseñado? Se sugieren cinco métricas significativas:

- Acoplamiento
- Cohesión
- Suficiencia
- Compleción (estado completo / plenitud)
- Ser primitivo

Acoplamiento: es una noción copiada del diseño estructurado, es la medida de la fuerza de la asociación establecida por una conexión entre un módulo 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. La complejidad puede reducirse diseñando sistemas con los acoplamientos más débiles posibles entre los módulos. Es igualmente importante el acoplamiento entre clases y objetos.

Existe tensión entre los conceptos de acoplamiento y herencia ya que la herencia implica un acoplamiento considerable. Es deseable clases débilmente acopladas pero es la herencia ayuda también a explotar las características comunes de las abstracciones.

Cohesión: también proviene del diseño estructurado. Mide el grado de conectividad entre los elementos de un solo módulo, clase u objeto. La forma de cohesión menos deseable es la cohesión por coincidencia, en la que se incluyen en la misma clase o módulo abstracciones sin ninguna relación. La más deseable es la cohesión funcional en la cual los elementos de una clase o módulo trabajan todos juntos para proporcionar algún comportamiento bien delimitado. Hay mayor cohesion cuando ay mas actividad entre ambos.

Suficiente: la clase o módulo captura suficientes características de la abstracción como para permitir una interacción significativa y eficiente.

Ser primitivo: Son aquellas q solo se pueden resolver cuando se tiene acceso a la implentacion

Estado completo: el interfaz de la clase o módulo captura todas las características significativas de la abstracción. Una clase o módulo completo es aquel cuyo interfaz es suficientemente general para ser utilizable de forma común por cualquier cliente. La compleción puede exagerarse. Ofrecer todas las operaciones significativas para una abstracción particular desborda al usuario y suele ser innecesario, ya que muchas operaciones de alto nivel pueden componerse partiendo de las de bajo nivel. Por ello se sugiere que las clases y módulos sean primitivos.

Operaciones primitivas: aquellas que pueden implantarse eficientemente sólo si tienen acceso a la representación subyacente. Una operación es primitiva si se puede implantar sólo accediendo a la representación interna. Una operación que podría implantarse sobre operaciones primitivas existentes pero a un coste de recursos computacionales significativamente mayor, es también candidata para su inclusión como operación primitiva.

Selección de operaciones.

Semántica funcional.

Es habitual en el desarrollo orientado a objetos diseñar los métodos de una clase como un todo, porque todos esos métodos cooperan para formar el protocolo completo de la abstracción. Así, dado un comportamiento que se desea hay que decidir en qué clase se sitúa. Para tomar una decisión de ese tipo se sugieren los siguientes criterios:

❑ **Reutilización**	¿Sería este el comportamiento más útil en más de un contexto?
❑ **Complejidad**	¿Qué grado de dificultad plantea el implementar este comportamiento?
❑ **Aplicabilidad**	¿Qué relevancia tiene este comportamiento para el tipo en el que podría ubicarse?
❑ **Conocimiento de la implementación**	¿Depende la implementación del comportamiento de los detalles internos de un cierto tipo?

Semántica espacial y temporal.

Una vez definida una operación y su semántica funcional hay que especificar la cantidad de tiempo que lleva completar la operación (semántica temporal) y la cantidad de espacio de almacenamiento que necesita (semántica espacial).

Siempre que un objeto pasa un mensaje a través de un enlace, ambos objetos deben estar sincronizados de alguna forma. En presencia de múltiples hilos de control esto significa que el paso de mensajes es mucho más que una mera selección de subprogramas. El paso de mensajes debe así adoptar una de las formas siguientes:

❖ **Síncrono**	Una operación comienza sólo cuando el emisor ha iniciado la acción y el receptor está preparado para aceptar el mensaje; el emisor y el receptor esperarán indefinidamente hasta que ambas partes estén preparadas para continuar.
❖ **Abandono inmediato**	Igual que el síncrono, excepto en que el emisor abandonará la operación si el receptor no está preparado inmediatamente.
❖ **De intervalo**	Igual que el síncrono, excepto en que el emisor esperará a que el receptor esté listo durante un intervalo de tiempo especificado.
❖ **Asíncrono**	Un emisor puede hincar una acción independientemente de si el receptor está esperando o no el mensaje.

Elección de relaciones.

Colaboraciones.

Define la accesibilidad. Por accesible se entiende la capacidad de una abstracción para ver a otra y hacer referencia a recursos en su vista externa. Una abstracción es accesible a otra sólo donde sus ámbitos se superpongan y sólo donde estén garantizados los derechos de acceso.

Los métodos de una clase no deberían depender de ninguna manera de la estructura de ninguna clase, salvo de la estructura inmediata (de nivel superior) de su propia clase.

Mecanismos y visibilidad.

La decisión sobre las relaciones entre objetos es principalmente una cuestión de diseñar los mecanismos por los que esos objetos interactúan. Durante el proceso de diseño es útil establecer explícitamente cómo un objeto es visible para otro. Existen cuatro formas fundamentales por las que un objeto **X** se puede hacer visible aun objeto **Y**:

- El objeto proveedor es global al cliente.
- El objeto proveedor es parámetro de alguna operación del cliente.
- El objeto proveedor es parte del objeto cliente.
- El objeto proveedor es un objeto declarado localmente en el ámbito del diagrama de objetos.

Hay una variación sobre cada una de estas ideas y es la idea de **visibilidad compartida**.

Elección de implementaciones.

Representación.

La representación de una clase u objeto debería casi siempre ser uno de los secretos encapsulados de la abstracción. Esto posibilita cambiar la representación sin violar ninguna de las suposiciones funcionales que los clientes puedan haber hecho.

A veces no es fácil elegir con que representación se queda el objeto (cuando dos o más se contradicen o compiten). De eso modo muchas veces se acaba con familias de clases cuyas interfaces son prácticamente idénticos, pero cuyas implantaciones son radicalmente distintas, con el fin de proporcionar diferentes comportamientos respecto al espacio y al tiempo.

Una de las compensaciones más difíciles cuando se selecciona la implantación de una clase se da entre el cálculo del valor del **estado de un objeto** y su **almacenamiento como un campo**.

Empaquetamiento.

Aparecen problemas similares en la declaración de clases y objetos dentro de los módulos.

Los requerimientos competidores de visibilidad y ocultación de información suelen guiar las decisiones de diseño sobre donde declarar clases y objetos. Generalmente, se busca construir módulos con cohesión funcional y débilmente acoplados. Hay muchos factores no técnicos que influyen estas decisiones como cuestiones de reutilización, seguridad y documentación. Al igual que el diseño de clases y objetos no hay que tomar a la ligera el diseño de módulos.

RESUMEN – CAPITULO 3

- ✓ Un objeto tiene estado, comportamiento e identidad.
- ✓ La estructura y comportamiento de objetos similares están definidos en su clase común.
- ✓ El estado de un objeto abarca todas las propiedades (normalmente estáticas) del mismo más los valores actuales (normalmente dinámicos) de cada una de esas propiedades.
- ✓ El comportamiento de la forma en que un objeto actúa y reacciona en términos de sus cambios de estado y paso de mensajes.
- ✓ La identidad es la propiedad de un objeto que lo distingue de todos los demás objetos.
- ✓ Los dos tipos de jerarquías de objetos son los lazos de **inclusión** y las relaciones de **agregación**.
- ✓ Una clase es un conjunto de objetos que comparten una estructura y comportamiento comunes.
- ✓ Los seis tipos de jerarquías de clase son las relaciones de asociación, herencia, agregación, “Uso”, instanciación y relaciones de metaclase.
- ✓ Las abstracciones clave son las clases y objetos que forman el vocabulario del dominio del problema.

Un mecanismo es una estructura por la que un conjunto de objetos trabajan juntos para ofrecer un comportamiento que satisfaga algún requerimiento del problema.

- ✓ La calidad de una abstracción puede medirse por su acoplamiento, cohesión, suficiencia, completad (estado completo), y por el grado hasta el cual es primitiva.

Capítulo 4 – Clasificación

La clasificación es el medio por el que ordenamos el conocimiento. En el diseño orientado a objetos, el renacimiento de la similitud entre las cosas nos permite exponer lo que tienen en común en abstracciones clave y mecanismos. No hay recetas fáciles para identificar clases y objetos. Existen técnicas de análisis orientado a objetos que ofrecen varias recomendaciones prácticas y reglas útiles para identificar las clases y objetos relevantes para un problema concreto.

4.1 La importancia de una clasificación correcta.

Clasificación y diseño orientado a objetos.

La identificación de clases y objetos es la parte más difícil del diseño OO. La experiencia muestra que la identificación implica descubrimiento e invención. Mediante el descubrimiento, se llega a reconocer las abstracciones clave y los mecanismos que forman el vocabulario del dominio del problema. Mediante la invención, se idean abstracciones generalizadas así como nuevos mecanismos que especifican como colaboran los objetos.
La clasificación también proporciona una guía para tomar decisiones sobre modularización. Se puede decidir ubicar ciertas clases y objetos juntos en el mismo módulo o en módulos diferentes, dependiendo de la similitud que se encuentra entre esas declaraciones, el acoplamiento y la cohesión son simplemente medidas de esta similitud.

La dificultad de la clasificación

Ejemplos de clasificación.

En el capítulo anterior se definió un objeto como algo que tiene una frontera definida con nitidez. Sin embargo, las fronteras que distinguen un objeto de otro son a menudo bastante difusas. Por ejemplo: ¿Dónde comienza la rodilla y donde termina?
Darwin propuso la teoría de que la selección natural fue el mecanismo de la evolución, en virtud de la cual las especies actuales evolucionaron de otras más antiguas. La teoría de Darwin dependía de una clasificación inteligente de especies.
Descartes afirmaba que "*el descubrimiento de un orden no es tarea fácil…, sin embargo, una vez que se ha descubierto el orden, no hay dificultad alguna en comprenderlo".*

La naturaleza incremental e iterativa de la clasificación.

La clasificación inteligente es un trabajo intelectualmente difícil, y la mejor forma de realizarlo es a través de un proceso incremental e iterativo. La naturaleza incremental e iterativa de la clasificación tiene un impacto directo en la construcción de jerarquías de clases y objetos en el diseño de un sistema de software complejo. Se puede dividir una clase grande en varias más pequeñas (composición). Se puede incluso descubrir aspectos comunes que habían pasado desapercibidos, e idear una nueva subclase (abstracción).

La dificultad radica en 2 razones importantes:

1. No existe algo que pueda llamarse una clasificación perfecta, algunas clasificaciones son mejores que otras. Potencialmente, hay al menos tantas formas de dividir el mundo en sistemas de objetos como científicos para emprender la tarea.
2. La clasificación inteligente requiere una gran perspicacia creativa: a veces la respuesta es evidente, a veces es una cuestión de gustos, y otras veces, la selección de componentes adecuados es un punto curial del análisis.

4.2 Identificando clases y objetos.

Enfoques clásicos y modernos.

Históricamente sólo han existido tres aproximaciones generales a la clasificación:

- ✓ Categorización clásica.
- ✓ Agrupamiento conceptual.
- ✓ Teoría de los prototipos.

Categorización clásica.

Es la aproximación clásica a la categorización: ***todas las entidades que tienen una determinada propiedad o colección de propiedades en común forman una categoría. Tales propiedades son necesarias y suficientes para definir la categoría.***

Un mayor conocimiento significativo sobre un dominio facilita, hasta cierto punto, conseguir una clasificación inteligente.

En sentido general, las propiedades pueden denotar algo más que meras características medibles; pueden también abarcar comportamientos observables. Por ejemplo: el hecho de que un pájaro pueda volar pero un pez no pueda, es una propiedad que distingue un águila de un salmón.

Agrupamiento conceptual

Es una variación más moderna del enfoque clásico. Representa más bien un agrupamiento probabilístico de los objetos. El agrupamiento conceptual está en estrecha relación con la teoría de conjuntos difusos (multivalor), en la que **los objetos pueden pertenecer a uno o más grupos**, en diversos grados de adecuación. Realiza juicios absolutos de la clasificación centrándose en la *mayor adecuación.*

Teoría de los prototipos

Existen abstracciones que no tienen ni propiedades ni conceptos delimitados con claridad. Por ejemplo, una categoría como juego no encaja en el molde clásico, porque no hay propiedades comunes compartidas por todos los juegos… tampoco hay una frontera fija para la categoría juego. Podría extenderse, se pueden introducir nuevos tipos de juegos, siempre que se parezcan a juegos anteriores de forma apropiada.
Esta es la razón por la que el enfoque se denomina *teoría de prototipos:* una clase de objetos se representa por un objeto prototípico, y se considera un objeto como un miembro de esta clase si y solo si se parece a este prototipo de forma significativa.

Aplicación de las teorías clásicas y modernas.

Las tres aproximaciones a la clasificación tienen aplicación directa en el diseño OO.
Se identifican las clases y objetos -en primer lugar- de acuerdo con las propiedades relevantes para nuestro dominio particular.

Análisis Orientado a Objetos

Las fronteras entre análisis y diseño son difusas. En el análisis se persigue modelar el mundo descubriendo las clases y objetos que forman el vocabulario del dominio del problema, y en el diseño se inventan las abstracciones y mecanismos que proporcionan el comportamiento que este modelo requiere.

Enfoques clásicos

Derivan sobre todo de los principios de la categorización clásica.

Ejemplos:
Shlaer y Mellor sugieren que las clases y objetos candidatos provienen habitualmente de una de las fuentes siguientes:

COSAS TANGIBLES	Coches, datos de telemetría, sensores de presión
PAPELES (ROLES)	Madre, profesor, político
EVENTOS	Aterrizaje, interrupción, petición
INTERACCIONES	Préstamo, reunión, intersección.

Desde la perspectiva del modelado de bases de datos:

PERSONAS	Humanos que llevan a cabo alguna función.
LUGARES	Areas reservadas para personas o cosas
COSAS	Objetos físicos, o grupos de objetos, que son tangibles.
ORGANIZACIONES	Colecciones formalmente organizadas de personas, recursos, instalaciones y posibilidades que tienen una misión definida.
EVENTOS	Cosas que suceden, habitualmente a alguna otra cosa, en una fecha y hora concretas, o como pasos dentro de una secuencia ordenada.

Coad y Yourdon sugieren otro conjunto más fuerte de objetos potenciales:

Estructuras	Relaciones “de clases” y “de partes”
Otros sistemas	Sistemas externos con los que la aplicación interactúa.
Dispositivos	Dispositivos con los que la aplicación interactúa.
Eventos recordados	Sucesos Históricos que hay que registrar.
Papeles desempeñados	Los diferentes papeles que juegan los usuarios en su interacción con la aplicación.
Posiciones	Ubicaciones físicas, oficinas y lugares importantes para la aplicación.
Unidades de organización	Grupos a los que pertenecen usuarios.

Análisis del comportamiento

Se centra en el comportamiento dinámico como fuente primaria de clases y objetos. Se forman clases basadas en grupos de objetos que exhiben comportamiento similar.
Las responsabilidades “denotan el conocimiento que un objeto tiene y las acciones que un objeto puede realizar. Las responsabilidades de un objeto son todos los servicios que suministra para todos los contratos que soporta. De esta forma se agrupan cosas que tienen responsabilidades comunes, y se forman jerarquías de clases que involucran a superclases que incorporan responsabilidades generales y subclases que especializan su comportamiento.
Al poner el énfasis en la comprensión de lo que sucede en el sistema, nos encontramos con los comportamientos del sistema, asignándolos a partes del sistema, donde surge la necesidad de entender quien inicia esos comportamientos y quien participa en ellos.
Los iniciadores y los participantes que desempeñan papeles significativos se reconocen como objetos, y se les asignan las responsabilidades de actuación para esos papeles.

Análisis de dominios.

Busca identificar las clases y objetos comunes a todas las aplicaciones dentro de un dominio dado, tales como mantenimiento de registros de pacientes, operaciones bursátiles, etc.
Se trata de un intento por identificar los objetos, operaciones y relaciones con los expertos del dominio consideran importantes acerca del mismo.

Análisis de Casos de Uso.

Una forma o patrón o ejemplo concreto de utilización, un escenario que comienza con algún usuario del sistema que inicia alguna transacción o secuencia de eventos interrelacionados.
Los usuarios finales, expertos del dominio y el equipo de desarrollo enumeran los escenarios fundamentales para el funcionamiento del sistema. Estos escenarios en conjunto describen las funcionales del sistema en esa aplicación. El análisis procede entonces como un estudio de cada escenario, utilizando técnicas de presentación.
A medida que el equipo pasa por cada escenario, debe identificar los objetos que participan en el, las responsabilidades de cada objeto, y cómo esos objetos colaboran con otros, en términos de las operaciones que invoca cada uno sobre el otro.

Fichas CRC

Las fichas CRC surgieron como una forma simple, pero maravillosamente efectivas de analizar escenarios. Es una tarjeta con una tabla de 3 X 5^5, sobre la cual el analista escribe el nombre de una clase (en la parte superior de la tarjeta), sus responsabilidades (en la mitad de la tarjeta) y sus colaboradores (en la otra mitad). Se crea una ficha para cada clase que se identifique como relevante para el escenario.
Pueden disponerse espacialmente para representar patrones de colaboración.

Descripción informal en español

Una alternativa al análisis OO clásico fue la propuesta de Abbott, quien sugirió redactar una descripción del problema (o parte del problema) en lenguaje natural y subrayar entonces los nombres y los verbos. Los nombres representan objetos candidatos, y los verbos representan operaciones candidatas sobre ellos. Este enfoque es útil porque es simple y porque obliga al desarrollador a trabajar en el vocabulario del espacio del problema.

Análisis Estructurado

Una segunda alternativa al Análisis OO clásico utiliza los productos del análisis estructurado como vía de entrada al diseño orientado a objetos. Esta técnica es atractiva sólo porque hay gran número de analistas con experiencia en análisis estructurado, y existen muchas herramientas CASE que soportan la automatización de estos métodos. Es desaconsejable el uso del análisis estructurado como punto de partida para el diseño OO.

4.3 Abstracciones y mecanismos clave.

Identificación de las abstracciones clave.

Búsqueda de las abstracciones clave

Una abstracción clave es una clase u objeto que forma parte del vocabulario del dominio del problema. El valor principal que tiene la identificación de tales abstracciones es que dan unos límites al problema; enfatizan las cosas que están en el sistema y, por lo tanto, son relevantes para el diseño y suprimen las cosas que están fura del sistema.
Por ejemplo: Un cliente que utiliza un cajero automático habla en términos de cuentas, depósitos y reintegros; estas palabras son parte del vocabulario del dominio del problema. Un desarrollador de un sistema semejante utiliza estas mismas abstracciones, pero introduce también algunas nuevas, como bases de datos, manejadores de pantallas, listas, cosas, etc. Estas abstracciones clave son artefactos del diseño particular, no del dominio del problema.

.

Refinamiento de abstracciones clave.

Una vez que se identifica determinada clave como candidata, hay que evaluarla de acuerdo a las mé6tricas descriptas en el capítulo anterior.
Dada una nueva abstracción, hay que ubicarla en el contexto de las jerarquías de clases y objetos que se han diseñado. En la práctica: No siempre se diseñan tipos en una jerarquía de tipos comenzando por un supertipo y creando a continuación los subtipos, sino que se crean varios tipos aparentemente dispares, se da cuenta de que están relacionados, y entonces factoriza sus características comunes en uno o más supertipos.
La colocación de clases y objetos en los niveles correctos de abstracción es difícil. A veces se puede encontrar una subclase general, y elegir moverla hacia arriba en la estructura de clases, incrementando así el grado de compartición. Esto se llama ***promoción de clases.***
Analógicamente se puede apreciar que una clase es demasiado general, dificultando así la herencia por las subclases a causa de un vacío semántico grande. Esto recibe el nombre de ***conflicto de granularidad.***

Sugerencias:

- ✓ **Los objetos** deberían **nombrarse** empleando frases construidas **con nombres propios**, como elSensor o simplemente forma.
- ✓ **Las clases** deberían **nombrarse** empleando frases construidas **con nombres comunes**, como Sensores o Formas.
- ✓ **Las operaciones de modificación** deberían nombrarse empleando **frases construidas con verbos activos,** como dibujar o moverIzquierda.
- ✓ **Las operaciones de selección** deberían implicar una interrogación, o bien nombrarse **con verbos del tipo "ser-estar",** como extensionDe o EstaAbierto.
- ✓ El uso de caracteres de subrayado y estilos de uso de mayúsculas son en gran medida cuestiones de gusto personal.

Identificación de mecanismos

Búsqueda de mecanismos.

Utilizamos la palabra mecanismo para describir cualquier estructura mediante la cual los objetos colaboran para proporcionar algún comportamiento que satisface un requerimiento del problema. Los mecanismos representan patrones de comportamiento.
El mecanismo que elige un desarrollador entre un conjunto de alternativas es frecuentemente el resultado de otros factores, como costo, fiabilidad, facilidad de fabricación y seguridad.
Una vez que el desarrollador decide sobre un patrón concreto de colaboración, se distribuye el trabajo entre muchos objetos definiendo métodos convenientes en las clases apropiadas.
Los mecanismos representan así decisiones de diseño estratégicas, como el diseño de una estructura de clases. La interfaz de una clase individual es más bien una decisión de diseño táctica.

Ejemplos de mecanismos

Consideremos el mecanismo de dibujo utilizado habitualmente en intercales gráficas de usuario. Varios objetos deben colaborar para presentar una imagen a un usuario: una ventana, una vista, el modelo que se va a visualizar, y algún cliente que sabe cuándo (pero no cómo) hay que visualizar ese modelo.

RESUMEN-CAPITULO 4

- ✓ La identificación de clases y objetos es el problema fundamental en el diseño orientado a objetos: la identificación implica descubrimiento e invención.
- ✓ La clasificación es fundamentalmente un problema de agrupación.
- ✓ La clasificación es un proceso incremental o iterativo, que se complica por el hecho de que un conjunto dado de objetos puede clasificarse de muchas formas igualmente correctas.
- ✓ Los tres enfoques de la clasificación son la **categorización clásica** (*clasificación por propiedades*), **agrupamiento conceptual** (*clasificación por conceptos*) y **teoría de prototipos** (*clasificación por asociación de un prototipo*)
- ✓ Los escenarios son una potente herramienta para el análisis orientado a objetos, y pueden utilizarse para guiar los procesos de análisis clásico, análisis del comportamiento y análisis de dominios.
- ✓ Las abstracciones clave reflejan el vocabulario del dominio del problema y pueden ser descubiertas en el dominio del problema, o bien ser inventadas como parte del diseño.
- ✓ Los mecanismos denotan decisiones estratégicas de diseño respecto a la actividad de colaboración entre muchos tipos diferentes de objetos.