Capítulo 4: Diseño Orientado a Objetos

Introducción

En este capítulo se describen tanto la noción de diseño para un sistema Orientado a Objetos como las principales actividades que corresponde realizar dentro de dicha etapa. Recordemos además que esta etapa es la segunda dentro de la metodología Orientada a Objetos, luego de la etapa de análisis (vista en el capítulo anterior) y previa a la etapa de implementación (que se verá en los capítulos siguientes).

La palabra *diseño* tiene una gran variedad de significados en diferentes contextos. En el contexto de la metodología Orientada a Objetos, podríamos decir que diseñar significa *tomar todas las decisiones relativas a los aspectos de implementación del Sistema y documentarlas apropiadamente*. La idea del diseño en términos de Orientación a Objetos consiste en generar un conjunto de documentos que le sirvan a los programadores para saber qué Clases deben implementar, qué métodos deben poseer y cómo deben comunicarse, entre otras cosas.

Es importante observar que los elementos mencionados en el párrafo anterior están fuertemente relacionados con aspectos propios de la implementación. Hablamos de Clases, pero a nivel del lenguaje de programación a utilizar (Java, para nosotros) y hablamos de métodos.

No hablamos de cosas como comprender los requerimientos o construir el Modelo de Análisis. Esas tareas ya fueron realizadas durante el análisis con el objetivo de comprender *qué* debía ser el Sistema a construir. En el diseño nos ocuparemos de decidir *cómo* implementaremos nuestro Sistema a efectos de que resuelva los requerimientos definidos durante el análisis.

Debe quedar bien claro porqué es importante destinar un tiempo prudencial al diseño. Muchos de problemas que surgen tras la implementación resultan de programar sin haber realizado un buen diseño. Por ejemplo, que el Sistema resultante se parezca poco a lo que se esperaba de él. Que tenga un gran número de *bugs* (errores en tiempo de ejecución). Que sea difícil extender el Sistema en uso para que contemple nuevos requerimientos, etc.

Si bien durante el análisis resolvimos el problema de decidir *qué* es lo que debe realizar el Sistema, todavía nos falta aprender elementos que nos ayuden a determinar *cómo* debemos programarlo. Dicho de otro modo, todavía no estamos en condiciones de sentarnos frente a la máquina y programar. Necesitamos aprender técnicas que nos permitan *organizar* la implementación, a efectos de evitar que la misma se haga de modo caótico y desordenado. De eso es que se ocupan las actividades a realizar durante la etapa de Diseño.

Actividades a realizar en la etapa de Diseño

Existen múltiples actividades a llevar a cabo durante la etapa de diseño. Todas ellas apuntan a definir el *cómo* se debe implementar el sistema. Las actividades concretas a llevar a cabo dependen de factores diversos, como ser características tecnológicas del sistema a construir o elementos propios de su *arquitectura* (temas a tratar en cursos posteriores). Dado que este es un *primer* curso de Orientación a Objetos vamos a restringir nuestras actividades de diseño solamente a las siguientes tres:

1. Elección de un diseño
2. Desglose de requerimientos
3. Especificación del diseño

Estas tres actividades serán suficientes para realizar nuestros diseños en el contexto del presente curso. No obstante, es importante recordar que existen múltiples actividades adicionales vinculadas al diseño.

Por último, así como en el Análisis fue importante contar con una notación clara y no ambigua para su realización, también será igualmente importante para el diseño. Nuevamente utilizaremos la notación UML, particularmente para la documentación de la tercera actividad (especificación del diseño). Para ello usaremos una variante al *Diagrama de Clases Conceptual* visto en el capítulo anterior, conocida como *Diagrama de Clases de Implementación*.

Elección de un diseño

Esta es la primera de las tres actividades a llevar a cabo durante la etapa de diseño. Consiste en realizar un *estudio de los Tipos Abstractos de Datos* que surgen en forma conjunta del *modelo de análisis* (Diagrama de Clases Conceptual) y del conjunto de *requerimientos* que el sistema deberá resolver. Dicho de otro modo, en esta actividad vamos a decidir qué Tipos Abstractos de Datos reflejan más apropiadamente la información plasmada en el modelo de análisis y permiten resolver en forma eficiente los requerimientos definidos para el Sistema.

Como se vio antes en el curso, desde una perspectiva formal las Clases de Objetos constituyen una *evolución* al concepto de Tipo Abstracto de Datos estudiado en el curso de Programación II. Lo que haremos en esta actividad será identificar los Tipos Abstractos de Datos que surgen de las clases plasmadas en el modelo de análisis.

Hay que tener presente que la elección de un diseño para un problema ***no necesariamente es único***. Para una ***misma*** realidad pueden existir ***varios*** diseños posibles. A partir de un mismo modelo de análisis (Diagrama de Clases Conceptual) pueden surgir distintas variantes de diseño, dependiendo de cuáles sean los *requerimientos* que el Sistema deba resolver. Incluso sucede que, para los mismos requerimientos, puede llegar a existir más de un diseño posible. En tal caso, a la hora de optar por una de dichas alternativas, pondremos en la balanza factores como la frecuencia de ejecución de cada requerimiento o el orden de los algoritmos involucrados. Rara vez una elección de diseño será tal que beneficie la eficiencia de todos los requerimientos. Generalmente optaremos por aquella alternativa que, en promedio, beneficie más al sistema como un todo.

Como dijimos antes, la elección de un diseño consiste en la realización de un estudio de Tipos Abstractos de Datos en base al modelo de análisis y los requerimientos definidos para el Sistema. A la hora de realizar dicho estudio se deben cumplir con los siguientes tres *criterios de trazabilidad* entre el análisis y el diseño.

1. **Consistencia con el Modelo de Análisis:** El estudio de Tipos Abstractos *debe ser consistente con las asociaciones y multiplicidades* del Diagrama de Clases Conceptual. Es decir, no deben existir contradicciones entre la forma de vinculación de las Clases a nivel del Diagrama y su correspondiente vinculación a nivel del diseño elegido.

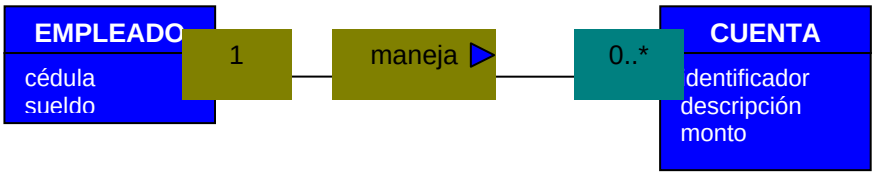
2. **Elección adecuada de Colecciones de Objetos:** Se deben elegir *colecciones de objetos* apropiadas para agrupar los objetos pertenecientes a las diferentes Clases presentes en el Diagrama conceptual. Para ello se debe consultar los *requerimientos* definidos para el Sistema y en base a ello elegir aquellos tipos *abstractos* de colecciones que, por sus características, resulten más adecuadas (*Diccionario*, *Secuencia*, *Queue*, etc.)

3. **Representación coherente de Asociaciones:** Así como a nivel del análisis, las Clases de vinculan mediante *asociaciones*, a nivel del diseño se vincularán a través de *nuevos atributos* que serán colocados en las distintas clases durante el estudio de Tipos Abstractos. Dichos nuevos atributos representarán *físicamente* las mismas asociaciones que a nivel del análisis se representaban *conceptualmente*.

Los tres criterios de trazabilidad apuntan a que ***no existan contradicciones*** entre la información dada por el análisis y su correspondiente representación a nivel de diseño. Por ejemplo, en el estudio de Tipos Abstractos no deberían surgir Colecciones de Objetos que no están presentes en el Diagrama Conceptual. Tampoco deberían vincularse físicamente dos clases que no poseen una línea de asociación que las une en el Diagrama Conceptual.

Ejemplo 1

Consideremos el siguiente modelo de análisis (Diagrama de Clases Conceptual) para una cierta realidad, junto con los requerimientos que se plantean luego del mismo.



1. Ingresar un nuevo empleado.
2. Dada la cédula de un empleado, asignarle una nueva cuenta. Su identificador será un número correlativo al último número de cuenta manejada por dicho empleado.
3. Dada la cédula de un empleado, sumar los montos de todas las cuentas manejadas por él.

Para esta realidad, vamos a proponer ***dos*** alternativas de diseño posibles y posteriormente optaremos por una de las dos.

1º alternativa: Presentamos el correspondiente *estudio de Tipos Abstractos de Datos*.

Empleados = Diccionario (Empleado).
Empleado = cédula x sueldo.
Cuentas = Secuencia (Cuenta).
Cuenta = identificador x descripción x monto x cédula.

En esta alternativa optamos por un *Diccionario* para almacenar a los Empleados debido a que puede usarse su cédula como clave y dos de los requerimientos exigen realizar búsquedas por cédula. Para el caso de las cuentas, se optó por una *Secuencia* debido a que no hay una clave única que la identifique y tampoco hay requerimientos que exijan búsqueda de Cuentas por clave. En este diseño hay una única colección que contiene a todos los empleados y una única colección que contiene a todas las cuentas. Para representar físicamente la asociación *maneja*, se incluyó un nuevo atributo a la clase Cuenta, que es la cédula del empleado que la maneja. Nótese que para un mismo empleado pueden existir muchas cuentas con su cédula (respetando así la multiplicidad 0..*), mientras que cada cuenta tiene solamente la cédula de un empleado (respetando así la multiplicidad 1).

2º alternativa: Presentamos el correspondiente *estudio de Tipos Abstractos de Datos*.

Empleados = Diccionario (Empleado).
Empleado = cédula x sueldo x CuentasEmp.
CuentasEmp = Secuencia (Cuenta).
Cuenta = identificador x descripción x monto.

En esta alternativa se observan las siguientes diferencias con respecto a la anterior. Ya no existe una única Secuencia que almacena a *todas* las cuentas del Sistema, sino que ahora a cada empleado se le asignó como atributo *su propia* Secuencia de Cuentas. Sí sigue existiendo un único Diccionario que alberga a todos los Empleados. Nótese también que cada cuenta ya no cuenta más con el atributo cédula del empleado que la maneja.

La vinculación ahora está dada del siguiente modo: Cada empleado posee todas sus cuentas manejadas en *su propia* Secuencia de cuentas (respetando así la multiplicidad 0..*), mientras que cada Cuenta sólo pertenece a la Secuencia de Cuentas del empleado que la maneja (respetando así la multiplicidad 1).

Estamos en presencia de ***dos*** alternativas de diseño válidas para la misma realidad y con los mismos requerimientos. Ambas respetan los tres criterios de trazabilidad mencionados antes. Sin embargo, la ***segunda*** alternativa es preferible, debido a las siguientes razones:

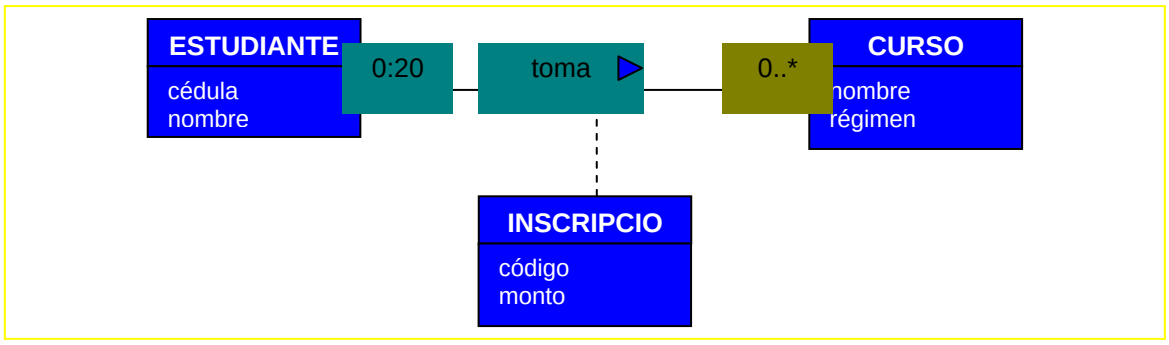
- El primer requerimiento (ingresar un nuevo empleado) es igualmente trabajoso en ambas alternativas, ya que consiste en insertar un nuevo Empleado en el (único) Diccionario de Empleados que existe.
- El segundo requerimiento (asignar una nueva cuenta a un empleado) se ve notoriamente beneficiado, ya que consiste únicamente en insertar una nueva cuenta en la secuencia de cuentas del empleado con un número mayor al largo de la secuencia. En la primera alternativa habría implicado recorrer *todas* las cuentas del sistema a efectos de determinar cuál era el último número de cuenta asignado al empleado.
- El tercer requerimiento (listar las cuentas manejadas por un empleado) también se ve notoriamente beneficiado debido que consiste en recorrer únicamente las cuentas de su secuencia de cuentas. En la primera alternativa habría implicado nuevamente recorrer *todas* las cuentas, filtrando aquellas cuya cédula coincide con la del empleado en cuestión.

Por regla general, aquellos diseños Orientados a Objetos que se basan en ***agregación*** (pertenencia en términos físicos) de objetos suelen ser preferibles a aquellos basados en ***vinculación por clave***. En la primera alternativa, se trabajó con esta última opción, vinculando a cada cuenta con su Empleado mediante la incorporación de su cédula (clave) como nuevo atributo. En cambio, la segunda alternativa trabajó con *agregación*, al incluir a cada Empleado *su propia* secuencia de cuentas como parte física del objeto, logrando aumentar notablemente la eficiencia en la resolución de los últimos dos requerimientos.

Ejemplo 2

A la hora de elegir un diseño, además de realizar el correspondiente *estudio de Tipos Abstractos de Datos*, también se deben especificar aquellas aclaraciones pertinentes que justifiquen el correcto cumplimiento de los tres *criterios de trazabilidad* vistos antes. Veamos otro ejemplo:

Dado el siguiente modelo de análisis (Diagrama de Clases Conceptual) para una realidad de una institución educativa, junto con los requerimientos que se plantean luego del mismo.



1. Inscripción de un nuevo estudiante.
2. Contar cuántos cursos siguen un determinado régimen.
3. Dado el código de una inscripción, obtener su monto junto con los datos del estudiante y del curso que le corresponden.

Elección de Diseño:

Estudiantes = Diccionario (Estudiante)
Cursos = Diccionario (Curso)
Inscripciones = Diccionario (Inscripción)
Estudiante = cédula x nombre
Curso = nombre x régimen
Inscripción = código x monto x Estudiante x Curso

Observaciones:

1. Hay una única colección de cada tipo en este diseño.
2. Se opta por Diccionarios para las tres colecciones debido a que cada una posee una clave que identifica a sus elementos, no pudiendo repetirse los mismos.
3. A cada inscripción se le asignan dos nuevos atributos: el estudiante y el curso correspondientes a dicha inscripción.
4. Un mismo estudiante estará referenciado desde muchas inscripciones, mientras que un mismo curso estará referenciado como máximo desde 20 inscripciones.

Estas observaciones explican de qué manera el diseño elegido respeta los tres criterios de trazabilidad y deben ser incluidas junto a toda elección de un diseño Orientado a Objetos.

Nótese que en este diseño ocurre que un mismo objeto puede estar referenciado *simultáneamente* desde varios lugares diferentes. Por ejemplo, un mismo Curso se encuentra simultáneamente en los siguientes lugares: Por un lado, formando parte del Diccionario que alberga a todos los cursos, y por el otro formando parte de hasta 20 inscripciones distintas. A la hora de diseñar sistemas Orientados a Objetos, pondremos énfasis en ***evitar la duplicación de objetos***. Es decir, procuraremos minimizar la duplicación de objetos, haciendo que un ***mismo*** objeto esté referenciado simultáneamente desde varios lugares distintos dentro de nuestra estructura de diseño.

Veremos más adelante que la forma de implementar esta simultaneidad vendrá dada por el uso de **referencias** (punteros) en la implementación. No obstante, dado que actualmente estamos en la etapa de *Diseño*, haremos nuestros estudios de Tipos Abstractos sin hablar expresamente de punteros ni cualquier otro aspecto vinculado a la etapa de implementación.

Limitaciones a la hora de elegir un buen diseño

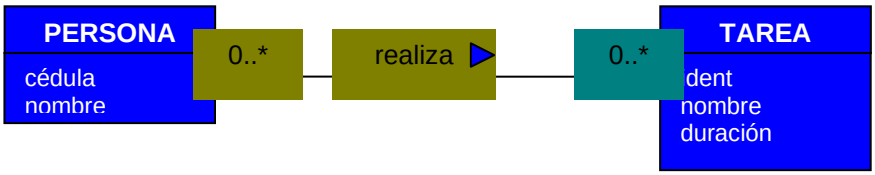
La acción de diseñar es una tarea compleja en múltiples sentidos, ya que se deben considerar muchos aspectos a la hora de realizar un buen diseño, y la experiencia ha demostrado que no siempre es posible optimizar absolutamente todos los aspectos vinculados. Queremos que el diseño elegido sea mantenible, que se adapte a los cambios, que sea eficiente para todos los requerimientos, que sea fácilmente extensible, reutilizable, etc.

Como se dijo anteriormente, rara vez una elección de diseño es tal que beneficia a todos los aspectos vinculados. Generalmente se opta por aquella alternativa que, en promedio, beneficia más al sistema como un todo, estudiando para cada sistema en concreto qué es lo que se considera más beneficioso para el sistema como un todo.

Muchas veces ocurre que el mejor diseño posible en términos de los requerimientos propuestos es muy engorroso de implementar, comparado a los beneficios reales que otorga en términos de eficiencia, extensibilidad o reusabilidad. Incluso hay ocasiones en las cuales el propio lenguaje de programación dificulta su implementación, como vemos en el siguiente ejemplo:

Ejemplo 3

Dado el siguiente modelo de análisis (Diagrama de Clases Conceptual), junto con los requerimientos que se plantean luego del mismo.



1. Dada la cédula de una persona, listar todas las tareas que realiza.
2. Dado el identificador de una tarea, listar todas las personas que la realizan.

El mejor diseño en términos de la eficiencia de los requerimientos propuestos sería éste:

Personas = Diccionario (Persona)
Persona = cédula x nombre x SecTareas
SecTareas = Secuencia (Tareas)

Tareas = Diccionario (Tarea)
Tarea = ident x nombre x duración x SecPer
SecPer = Secuencia (Persona)

En este diseño, cada Persona cuenta con su propia Secuencia de Tareas realizadas (especialmente útil para el primer requerimiento), y a su vez cada Tarea cuenta con su propia Secuencia de las Personas que la realizan (especialmente útil para el segundo requerimiento). En términos de la eficiencia de cada requerimiento por sí solo, este sería sin dudas el mejor diseño.

Sin embargo, es un diseño que presenta diversas complicaciones. En primer lugar nótese la cantidad de colecciones necesarias para su implementación. Además de los Diccionarios para almacenar todas las Personas y Tareas del sistema, se suma una secuencia de Tareas para cada Persona y una Secuencia de Personas para cada Tarea. Si bien se cumple con el principio de no duplicación de objetos (dado que cada objeto pertenece simultáneamente a muchas colecciones), no es difícil percibir que se trata de una estructura de objetos demasiado compleja solamente para dos requerimientos. El trabajo de mantener dicha estructura en la implementación quizás resulte mucho mayor que los beneficios reales de eficiencia que aporta.

En base a todas estas consideraciones, la moraleja parece ser nuevamente que la elección de un diseño debe ser tal que *beneficie promedialmente al sistema como un todo*. Por esta razón, proponemos un diseño alternativo para este ejemplo que, si bien quizás no sea el más eficiente en términos de los requerimientos propuestos, resultará más *mantenible*, *extensible* y por sobre todas las cosas *factible de ser implementado*.

Diseño alternativo:

Personas = Diccionario (Persona)
Tareas = Diccionario (Tarea)
Parejas = Secuencia (Pareja)
Pareja = Persona x Tarea
Persona = cédula x nombre
Tarea = ident x nombre x duración

Observaciones:

1. Hay una única colección de cada tipo en este diseño.
2. Se opta por Diccionarios para las colecciones de Personas y Tareas debido a que cada una posee una clave que identifica a sus elementos, no pudiendo repetirse los mismos.
3. Se crea una nueva clase Pareja que vincule a una Persona con una Tarea realizada.

4. Todas las parejas se almacenan en una Secuencia, puesto que sólo se debe listarlas.
5. Cada Persona estará referenciada simultáneamente desde el Diccionario de Personas y desde muchas Parejas distintas. Lo mismo ocurre con las Tareas.

Nótese que en este diseño no solamente se realizaron vinculaciones físicas mediante atributos para representar las asociaciones del modelo de análisis, sino que además se creó una clase nueva (Pareja) que no estaba presente en el análisis, pero que sirvió como alternativa para simplificar el diseño. La incorporación de nuevas clases a un diseño es válida siempre y cuando se haga de forma tal que se sigan cumpliendo los tres *criterios de trazabilidad* entre análisis y diseño.

Desglose de Requerimientos

Esta es la segunda de las tres actividades a llevar a cabo durante la etapa de diseño. Como su nombre lo indica, consiste en descomponer (*desglosar*) cada requerimiento propuesto a efectos de definir los algoritmos que conducirán a su resolución. El desglose de requerimientos será la fuente que luego permitirá definir las operaciones (los métodos) a incorporar en las clases a la hora de implementar, de modo tal que se puedan resolver los requerimientos.

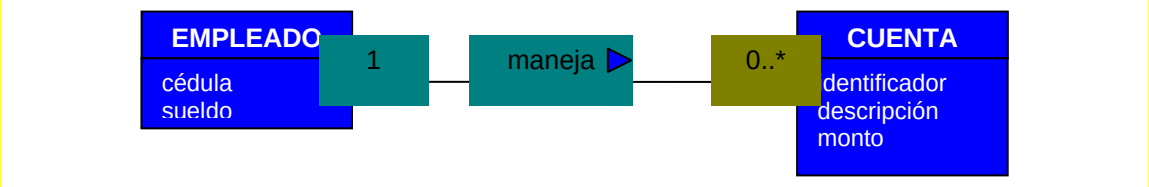
Existen diversas notaciones para realizar esta segunda actividad de diseño. La propia notación UML provee unos diagramas especiales (denominados *diagramas dinámicos*) definidos expresamente para realizar esta actividad. No obstante, debido a que el nivel de complejidad de nuestros diseños no será elevado en este primer curso de Orientación a Objetos, nos bastará simplemente con utilizar la notación de *seudocódigo* para definición de algoritmos vista en el curso de Programación I.

Por ***cada*** requerimiento definido para el Sistema habremos de realizar su correspondiente desglose. El desglose debe partir de la elección de diseño realizada para el Sistema y de los desarrollos expandidos de los casos de uso, y tomarlos en cuenta a la hora de definir la forma en la cual el requerimiento será resuelto con esa estructura de objetos. La profundidad del desglose debe ser tal que permita definir claramente la forma en la cual vamos a *recorrer* nuestra estructura de objetos, pero sin brindar detalles específicos de implementación (como ser, por ejemplo, las estructuras concretas de datos a utilizar), los cuales se definirán posteriormente dentro del proceso de construcción del Sistema.

Presentamos a continuación ejemplos de desglose de requerimientos.

Ejemplo 4

Consideremos nuevamente el Diagrama de Clases Conceptual y los requerimientos del ejemplo 1 presentado anteriormente.



1. Ingresar un nuevo empleado.
2. Dada la cédula de un empleado, asignarle una nueva cuenta. Su identificador será un número correlativo al último número de cuenta manejada por dicho empleado.
3. Dada la cédula de un empleado, sumar los montos de todas las cuentas manejadas por él.

Para este ejemplo, tomaremos en consideración la segunda alternativa de diseño oportunamente definida para esta realidad:

Empleados = Diccionario (Empleado).
Empleado = cédula x sueldo x CuentasEmp.
CuentasEmp = Secuencia (Cuenta).
Cuenta = identificador x descripción x monto.

Vamos a desglosar a continuación cada uno de los tres requerimientos, para lo cual consultamos el ***desarrollo expandido*** del caso de uso correspondiente a cada uno.

Requerimiento 1: Ingresar un nuevo empleado.

Mostramos aquí el desarrollo expandido que los analistas de requerimientos hicieron para el caso de uso correspondiente a este requerimiento:

Nombre del caso de uso:	Ingresar nuevo empleado
Actores:	Funcionario de la empresa
Puntos de extensión:	Ninguno
Puntos de inclusión:	Ninguno
Extiende a:	Ninguno
Incluye a:	Ninguno
Precondiciones:	Ninguna
Poscondiciones:	Ingreso realizado
Flujo principal:	1. Sistema solicita datos del nuevo empleado 2. Usuario ingresa los datos 3. Sistema verifica si ya estaba ingresado 4. Si lo estaba, ir al flujo alterno [A1] 5. Si no lo estaba, registrarlo en el sistema.
Flujo alternativo A1:	1. Sistema emite mensaje de error y termina el caso de uso.

Este desarrollo expandido nos muestra el comportamiento del requerimiento desde la prespectiva del usuario. Lo que vamos a hacer es desglosar el requerimiento en términos de su comportamiento a nivel de la estructura de objetos del diseño elegido. Dicho de otro modo, seguiremos mostrando cómo se comporta el requerimiento, pero en vez de hacerlo desde la perspectiva del usuario, lo haremos desde la perspectiva de lo que sucede *adentro* del sistema.

Desglose:

A partir de la cédula, verificar si el empleado pertenece al Diccionario de Empleados.
Si ya pertenece **entonces**
    Error: ya existe un empleado con dicha cédula en el sistema.
Sino
    Insertar el nuevo objeto Empleado en el Diccionario de Empleados.
Fin

Obsérvese que el comportamiento del caso de uso sigue siendo el mismo que el del desarrollo expandido, pero esta vez está expresado en términos de lo que sucede en forma interna a los objetos del sistema, lo que posteriormente nos será de utilidad para definir los métodos que habremos de implementar en las distintas clases.

Requerimiento 2: Dada la cédula de un empleado, asignarle una nueva cuenta. Su identificador será un número correlativo al último número de cuenta manejada por dicho empleado.

Mostramos aquí el desarrollo expandido que los analistas de requerimientos hicieron para el caso de uso correspondiente a este requerimiento:

Nombre del caso de uso:	Asignar cuenta a empleado
Actores:	Funcionario de la empresa
Puntos de extensión:	Ninguno
Puntos de inclusión:	Ninguno
Extiende a:	Ninguno
Incluye a:	Ninguno
Precondiciones:	Ninguna
Poscondiciones:	Cuenta asignada
Flujo principal:	1. Sistema solicita la cédula del empleado y datos de la nueva cuenta a asignar 2. Usuario ingresa los datos 3. Sistema verifica si el empleado existe en el sistema 4. Si no existe, ir al flujo alterno [A1] 5. Si existe, asignarle a la cuenta el número siguiente al de la última cuenta del empleado y registrarla en el sistema.
Flujo alternativo A1:	1. Sistema emite mensaje de error y termina el caso de uso.

Mostramos ahora el desglose correspondiente:

Desglose:

A partir de la cédula del empleado, verificar si pertenece al Diccionario de Empleados.
Si pertenece **entonces**
Obtener el objeto Empleado del Diccionario de Empleados.
cant = cuántos objetos posee la Secuencia de Cuentas del Empleado.
Asignar identificador = cant + 1 al nuevo objeto Cuenta.
Acceder a la Secuencia de Cuentas del Empleado e insertarle la nueva cuenta.
Sino
Error: no existe el empleado buscado.
Fin

Requerimiento 3: Dada la cédula de un empleado, sumar los montos de todas sus cuentas.
Mostramos el desarrollo expandido correspondiente a este requerimiento:

Nombre del caso de uso:	Monto total de cuentas
Actores:	Funcionario de la empresa
Puntos de extensión:	Ninguno
Puntos de inclusión:	Ninguno
Extiende a:	Ninguno
Incluye a:	Ninguno
Precondiciones:	Ninguna
Poscondiciones:	Monto calculado
Flujo principal:	1. Sistema solicita la cédula del empleado. 2. Usuario la ingresa. 3. Sistema verifica si el empleado existe en el sistema 4. Si no existe, ir al flujo alterno [A1] 5. Si existe, calcular la suma de los montos de sus cuentas.

Flujo alternativo A1:	1. Sistema emite mensaje de error y termina el caso de uso.

Mostramos ahora el desglose correspondiente:

Desglose:

A partir de la cédula del empleado, verificar si pertenece al Diccionario de Empleados.
Si pertenece **entonces**
  Acceder a la Secuencia de Cuentas del Empleado.
  Para cada Cuenta de la Secuencia **hacer**
    Acumular su monto.
  Fin
Sino
  Error: no existe el empleado buscado.
Fin

Nótese que en ninguno de estos tres desgloses se dieron detalles específicos de la implementación, ya que no es el objetivo hacerlo en este punto. Por ejemplo, no se dijo cual es la estructura de datos utilizada para el Diccionario (Hash, ABB, etc.). Tampoco se dijo si la inserción de la nueva cuenta se hace al principio o al final de la estructura de datos que implementa la secuencia. Tales aspectos serán definidos más sobre la etapa de implementación.

Lo que interesa dejar plasmado en el desglose de cada requerimiento son las operaciones que interesa realizar sobre las colecciones y sobre los objetos, como forma de validar que los requerimientos son resolubles en base al diseño elegido. Por ejemplo, interesa dejar constancia de que se debe *buscar* un objeto en el diccionario, pero sin especificar si la búsqueda será en un Hash o un ABB. Interesa dejar constancia de que se debe recorrer la Secuencia de cuentas, pero sin especificar la estructura de control a utilizar en la recorrida o la estructura de datos de la secuencia.

El desglose de los requerimientos debe contener un nivel de detalle tal que posteriormente permita definir cuáles serán los encabezados de las operaciones (los *métodos*) a ser incorporados en cada clase. La definición de tales encabezados, así como de otros aspectos cada vez más cercanos a la implementación serán definidos en la 3º y última actividad dentro del diseño: ***La especificación***.

Especificación del Diseño

Esta es la tercera y última de las tres actividades a llevar a cabo durante la etapa de diseño. Como su nombre lo indica, consiste en *especificar* detalladamente todas las decisiones de diseño realizadas, con la finalidad de generar una *documentación* clara y (en lo posible) libre de ambigüedades que posteriormente sirva como guía para la etapa de implementación. Cuanto más detallada sea la especificación, menos posibilidades existirán de cometer errores al implementar.

La actividad de especificación se compone, a su vez, de dos sub-actividades:

1. Creación del *Diagrama de Clases de Implementación* (a partir del diseño elegido).
2. Incorporación de *métodos* a las clases del diagrama (a partir del desglose de requerimientos).

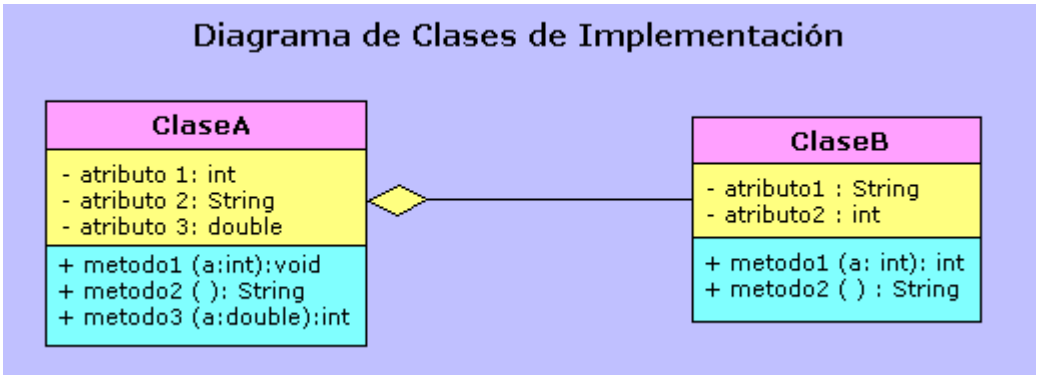
Cuando realizamos la elección del diseño, nos ocupamos de realizar un *Estudio de tipos Abstractos de Datos* que fuera consistente con el *Diagrama de Clases Conceptual* (Modelo de Análisis) y acorde a los requerimientos propuestos para el sistema. Lo hicimos de modo tal que se cumplieran los tres *criterios de trazabilidad* entre el análisis y el diseño. En la especificación del diseño, nos ocuparemos de traducir la elección de diseño realizada en un nuevo Diagrama de Clases, llamado *Diagrama de Clases de Implementación*.

El Diagrama de Clases de Implementación para un Sistema contendrá toda la información necesaria para que el programador sepa cuáles son las Clases que debe implementar, qué atributos deberán poseer (con sus correspondientes tipos) y cuáles serán los cabezales de los métodos que deben brindar (con sus correspondientes parámetros y tipos de retorno). La idea es que *mirando* el Diagrama de Clases, el programador sabrá exactamente cómo debe ser el "esqueleto" de cada una de las Clases que se deben implementar.

La incorporación de métodos a las clases de este nuevo diagrama se hará en base al *desglose de requerimientos*, el cual representará la fuente de la cual surgirán los métodos que es necesario incorporar a las clases, de modo tal que los requerimientos del sistema sean resueltos tal y como se indicó en el desglose.

Diagrama de Clases de Implementación

Este diagrama se construye *traduciendo* el diseño elegido (estudio de Tipos Abstractos de Datos) a un formato gráfico (el nuevo diagrama), el cual estará formado por Clases y *únicamente* asociaciones de *agregación*, *composición* y/o *herencia*.



No habrá en este diagrama otros tipos de asociaciones aparte de estos tres. La razón para ello es que todas las asociaciones del Diagrama Conceptual fueron traducidas a nuevos atributos que las representan *físicamente* en el contexto del diseño elegido.

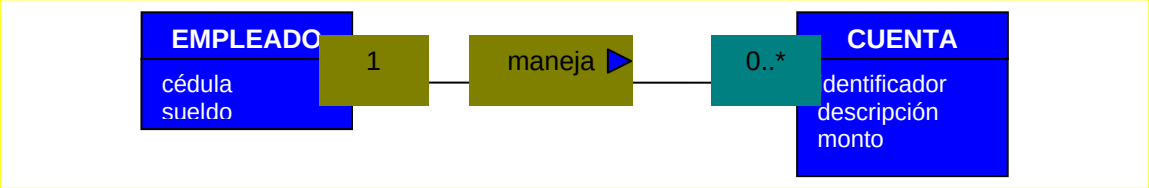
Cada clase existente en el diseño elegido (tanto aquellas que ya venían del Diagrama Conceptual, como aquellas que representan Colecciones de Objetos u otras clases introducidas en la elección del diseño), será representada en este nuevo diagrama. Los nuevos atributos incorporados para representar físicamente las asociaciones serán representados mediante *agregaciones*. Las únicas asociaciones que permanecerán incambiadas respecto al Diagrama Conceptual son las relaciones de composición y herencia que ya pudieran existir. Como resultado, el Diagrama de Clases de Implementación será un fiel reflejo del diseño elegido.

Es importante recordar que ambos diagramas (conceptual y de implementación) tienen *significados* sustancialmente diferentes. El Diagrama *Conceptual* es una forma gráfica de representar al *Modelo de Análisis* del cual tanto hemos hablado, mientras que el Diagrama de *Implementación* le servirá al programador para sentarse frente a la máquina y codificar.

Veamos un primer ejemplo introductorio que ilustra la forma en la cual se construye este diagrama:

Ejemplo 5

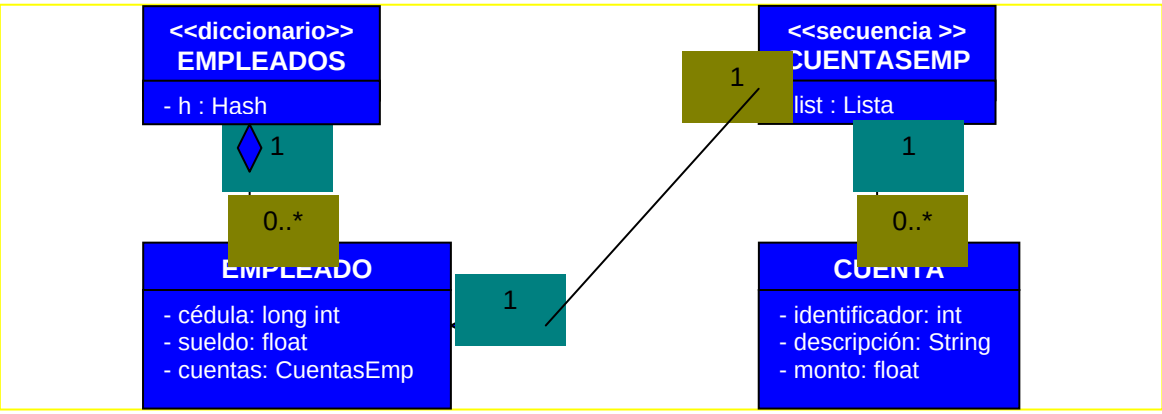
Consideremos nuevamente el Diagrama de Clases Conceptual del ejemplo 4 y su correspondiente elección de diseño:



El diseño elegido para esta realidad fue el siguiente:

Empleados = Diccionario (Empleado).
Empleado = cédula x sueldo x CuentasEmp.
CuentasEmp = Secuencia (Cuenta).
Cuenta = identificador x descripción x monto.

Vamos a dibujar el correspondiente Diagrama de Clases de Implementación:



Este Diagrama de Clases de Implementación representa fielmente el diseño elegido y cumple con los criterios de trazabilidad. Hay un *único* Diccionario que contiene a todos los Empleados. Cada Empleado contiene *su propia* Secuencia de Cuentas (véase que es un nuevo atributo que le pusimos a la clase Empleado) y a su vez cada objeto de este diseño (Empleado o Cuenta) está almacenado en una única colección, respetándose las multiplicidades provenientes del análisis.

Nótese que las tres relaciones presentes son de *agregación*, la cual sabemos que representa una relación de *pertenencia en términos físicos*. Todos los empleados del sistema están físicamente almacenados en el diccionario de empleados (*agregación* entre las Clases **Empleados** y **Empleado**). A su vez cada empleado contiene su propia secuencia de cuentas (agregación entre las clases **Empleado** y **CuentasEmp**) y por último, cada cuenta está físicamente almacenada dentro de una secuencia de cuentas (agregación entre las clases **CuentasEmp** y **Cuenta**).

Obsérvese además que ya hemos incorporado los primeros detalles de implementación al diagrama, indicando de qué *tipo* serán los atributos al implementar y qué *estructuras de datos* se utilizarán para las colecciones de este diseño. Por ejemplo, hemos decidido que la cédula de cada empleado sea de tipo *long int* y que el identificador de cada cuenta sea de tipo *int*.

Con respecto a las colecciones, hemos decidido que el Diccionario de Empleados se represente internamente con un *Hash*, mientras que cada Secuencia de Cuentas de represente internamente con una *Lista*. La elección de las estructuras de datos responde a cuestiones de *eficiencia*, *cardinalidad* y *frecuencia* de las operaciones. Por ejemplo, para el Diccionario se eligió un *Hash* porque es lo más adecuado para búsquedas por clave (requerimientos 1, 2, 3) y no es acotado. Para las secuencias de cuentas se eligió una *Lista* porque no hay cota y alcanza con mantener a las cuentas almacenadas en forma lineal.

Por último, todos los atributos se marcaron con un signo de menos (-), que en notación UML significa que los atributos serán ***privados*** dentro de las clases y no podrán ser accedidos directamente desde fuera. Esto garantizará el cumplimiento de la propiedad de *encapsulamiento*.

Este Diagrama de Clases de Implementación aún **no está completo**. Para completarlo, habrá que incorporarle *métodos* a las clases, lo cual veremos a continuación.

Incorporación de Métodos al Diagrama

Para completar el Diagrama de Clases de Implementación, vamos a incorporarle los *cabezales* de todos los métodos que sean necesarios para resolver los requerimientos del sistema. La fuente para la incorporación de métodos será el *desglose de requerimientos* realizado en base al diseño elegido. Cuanto más detallado sea el desglose, más fácil será definir qué métodos es necesario incorporar a cada una de las clases.

Hay dos categorías de métodos a incorporar en las Clases del Diagrama:

1. Métodos ***primitivos***. Se trata de las operaciones primitivas del T.A.D correspondiente a cada clase del diagrama. Por regla general, se suelen incorporar todas las operaciones primitivas del T.A.D que corresponde a cada clase, aunque puede haber diseños en los cuales optemos por incluir solamente aquellas primitivas estrictamente necesarias para nuestros requerimientos.
2. Métodos ***específicos***. Se trata de operaciones que **no** son primitivas y que son necesarias para la resolución de los requerimientos específicos del diseño en cuestión. Puede haber requerimientos que se resuelvan exclusivamente invocando a las operaciones primitivas, en esta categoría caen todos aquellos requerimientos que no son resolubles utilizando únicamente primitivas o bien aquellos que, aún pudiendo resolverse con primitivas, se incorporan como métodos auxiliares que ayudan a repartir mejor el trabajo entre las clases.

Los métodos a incorporar serán ***públicos*** dentro de las clases, lo que significa que podrán ser invocados desde *fuera* de las clases, y serán los únicos que tendrán permitido acceder directamente a los atributos ***privados*** de la clase a la que pertenecen. Los métodos serán incluidos al diagrama en un segundo recuadro, por debajo de los atributos. La notación UML para indicar que son públicos es un signo de mas (+).

En el Diagrama de Clases de Implementación, tenemos presentes dos tipos de clases. Por un lado las *Clases Básicas*, que representan objetos de la realidad obtenidas originalmente del *Modelo de Análisis* junto con las *clases de asociación* que, si bien no representan objetos "reales", cumplen en la práctica la misma función. Por otro lado tenemos las clases que representan *Colecciones de Objetos*, las cuales fueron introducidas durante la elección del diseño.

Existen tres tipos de métodos primitivos que deben estar presentes en las Clases Básicas. Los mismos se denominan *Constructores*, *Selectores* y *Modificadores*. Los métodos constructores permiten construir nuevos objetos de la Clase a la cual pertenecen. Poseen el mismo nombre que la clase a la cual pertenecen y reciben como parámetros aquellos valores con los cuales van a ser inicializados los atributos del objeto al momento de su creación. Una clase puede poseer más de un método constructor, pero siempre es recomendable que posea al menos uno, el cual debería recibir suficientes parámetros como para inicializar todos sus atributos.

Los métodos selectores son las funciones que permiten consultar los valores almacenados en cada uno de los atributos del objeto sobre el cual se aplican, mientras que los métodos modificadores son procedimientos que permiten modificar los valores almacenados en cada uno de los atributos del objeto sobre el cual se aplican.

Con respecto a las Colecciones de Objetos, los métodos primitivos que deben estar presentes dependerán del Tipo Abstracto de Datos al cual pertenezca la colección, la cual puede ser un *Diccionario*, una *Secuencia*, un *Stack*, un *Set*, etc. Además de ellas, cada Colección debe contar también con un método constructor, el cual no recibe ningún parámetro la mayoría de las veces, y simplemente se encarga de crear la colección vacía. No obstante, para realidades en las cuales no tenga sentido que la colección esté vacía, se le puede pasar como parámetro el o los objetos con los cuales vaya a ser inicializada.

En relación a los métodos específicos, como se dijo anteriormente se trata de métodos que no forman parte de los primitivos y que se introducen con la finalidad de resolver requerimientos específicos del diseño en cuestión, o bien métodos adicionales que facilitan la resolución de los mismos o bien reparten más adecuadamente el trabajo entre las diferentes clases. Estos métodos surgen específicamente del *desglose de requerimientos*, particularmente en lo relativo a la ejecución de pasos tales que no se pueden resolver exclusivamente con métodos primitivos.

La sintaxis UML para escribir el cabezal de un método cualquiera es la siguiente:

```
nombreMetodo (tipoParametro1, ... ,tipoParametroN) : tipoRetorno
```

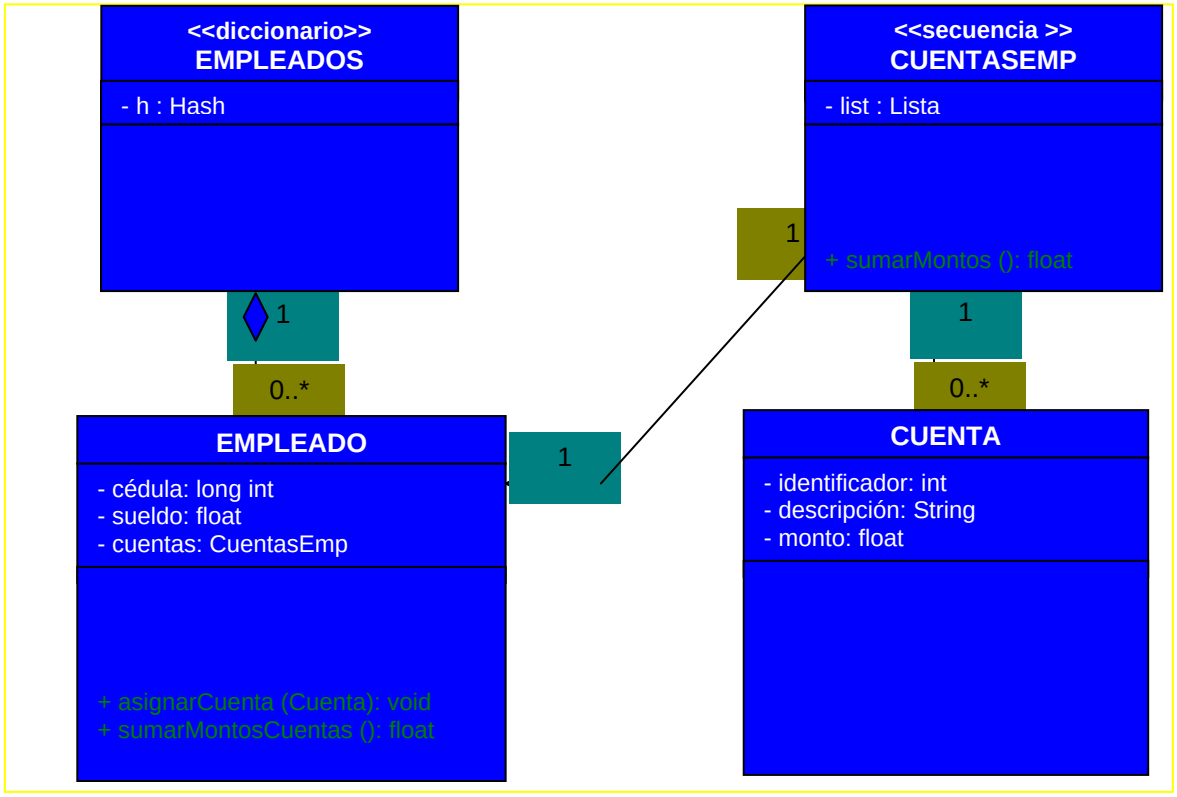
En los Lenguajes Orientados a Objetos, todos los métodos (tanto primitivos como específicos) **nunca** reciben como parámetro el objeto sobre el cual se aplican, ya que el mismo es pasado implícitamente como parámetro al momento de su invocación. Solamente se pasan como parámetros aquellos valores que expresamente deben ser recibidos desde el exterior. Así, por ejemplo, el método **modificarDomicilio** de una Clase **Persona** tendrá el siguiente cabezal:

```
modificarDomicilio (String) : void
```

Obsérvese que, en Programación estructurada, este método habría tenido además a la persona como primer parámetro, además del domicilio a modificar. En Orientación a Objetos, la persona es pasada implícitamente, por lo que no se incorpora en el cabezal.

Ejemplo 6

Vamos a incorporar métodos al Diagrama del ejemplo 5. Mostramos en **azul** los métodos primitivos y en **verde** los métodos específicos en cada clase:



En relación a los métodos primitivos, en la Clase **Empleados** se incorporaron todas las primitivas habituales del T.A.D *Diccionario*, mientras que en la Clase **CuentasEmp** solamente se incorporaron aquellas primitivas del T.A.D *Secuencia* que se juzgaron necesarias para satisfacer los requerimientos del sistema. En cuanto a las clases básicas **Empleado** y **Cuenta**, se incorporó el constructor, todas las selectoras y solamente aquellos métodos modificadores que se juzgaron necesarios. La decisión de incorporar o no todas las primitivas correspondientes a cada Clase es una decisión que depende de cada diseño en particular.

Con respecto a los métodos específicos, no se incorporó ninguno a la Clase **Empleados** porque en base al desglose de los requerimientos 1, 2 y 3, todas las acciones realizadas sobre el diccionario podían resolverse exclusivamente utilizando los métodos primitivos. En cambio, en la clase CuentasEmp se incorporó el método **sumarMontos** que no pertenece a las primitivas del T.A.D *Secuencia*, pero que es apropiado para calcular la suma de montos del requerimiento 3.

Obsérvese que **no** se incorporaron los métodos **getCuentas** y **setCuentas** a la clase **Empleado**, en su lugar se incorporaron los métodos específicos **asignarCuenta** y **sumarMontosCuentas** que internamente invocarán a los métodos de la Secuencia de Cuentas. Esto ayuda a aumentar aún más el *encapsulamiento* y repartir mejor el trabajo entre las distintas clases.

En Orientación a Objetos, cuando una Clase posee una *colección* entre sus atributos (como ocurre entre **Empleado** y **CuentasEmp**), es una buena práctica que la Clase se haga responsable de encapsular *totalmente* el acceso a su Colección, evitando que la misma pueda ser accedida directamente desde el exterior, proveyendo en su lugar métodos auxiliares (tales como **asignarCuenta** y **sumarMontosCuentas**) que oficien de intermediarios.

Herencia y Polimorfismo en Diseño

Sabemos que la herencia define una relación *"ser"* entre dos Clases de Objetos, las cuales se denominan *clase base* (o *superclase*) y *clase derivada* (o *subclase*). Esta relación significa que todos los objetos de la clase derivada *son* también objetos de la clase base. Del análisis sabemos que la clase derivada hereda **todos** los atributos de su clase base, además de poder poseer atributos propios. Veremos en esta sección que la clase derivada también hereda los **métodos** de su clase base, con excepción de los métodos constructores.

La clase derivada debe poseer sus propios métodos constructores, los cuales deben recibir suficientes parámetros para poder inicializar todos sus atributos, tanto aquellos heredados de la clase base como aquellos propios definidos en ella. Veremos más adelante, en la etapa de implementación, que los constructores de las clases derivadas internamente invocarán a alguno de los constructores de su clase base.

Hay una categoría especial de métodos específicos que puede poseer una clase derivada. Se trata de los **métodos polimórficos**, Un método polimórfico es aquel que está originalmente implementado en la Clase base y que luego es re-definido o implementado nuevamente en alguna clase derivada. A este mecanismo se lo conoce también como *sobre-escritura* de métodos. Hay situaciones en las cuales se desea que una determinada operación, ya existente en la clase base, se comporte en forma diferente en la clase derivada, o bien que extienda el comportamiento previamente definido en la clase base. En UML, para representar que un método es polimórfico (o sea, que está siendo sobre-escrito en una clase derivada) lo que se hace es *repetir* el cabezal del método en la clase derivada.

La invocación de un método polimórfico en Orientación a Objetos no requiere preguntar previamente de qué tipo es el Objeto a la hora de decidir cuál de las dos versiones del método ejecutar. Veremos que en la implementación se resuelve automáticamente en tiempo de ejecución.

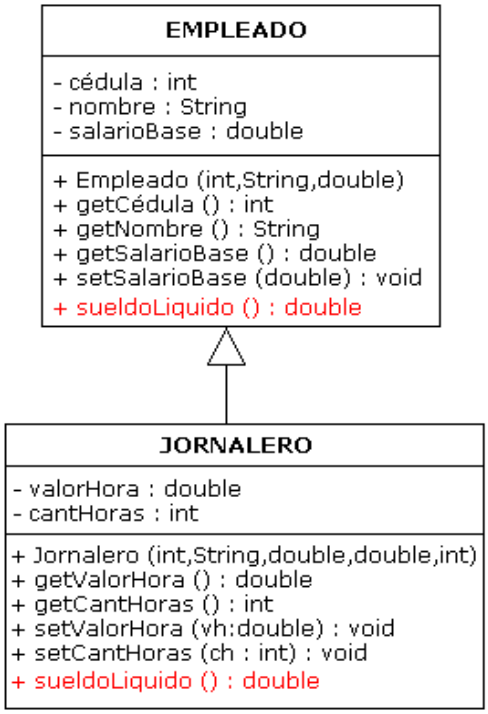
Ejemplo 7

Dado el siguiente fragmento de un Diagrama de Clases de Implementación:

El constructor de la clase base recibe tres parámetros, mientras que el constructor de la clase derivada recibe cinco (los tres anteriores más los dos propios). Con excepción del constructor, la clase derivada heredó **todos** los atributos y métodos de la clase base.

Se ha incorporado además un método polimórfico denominado **sueldoLiquido** que calcula el sueldo líquido de un empleado cualquiera, ya sea común o jornalero. Si bien *ambos* tipos de empleado tienen dicho método, el comportamiento dependerá del tipo de empleado en cuestión.

Por ello, la clase Jornalero incorpora nuevamente el cabezal de dicho método, ya que el comportamiento del mismo (heredado de la clase base) no es adecuado para un jornalero, puesto que en el cálculo deben intervenir su valor hora y su cantidad de horas trabajadas, además del salario base. Nótese que finalmente, la clase Jornalero cuenta en total con 10 métodos (cuatro heredados + su constructor + cuatro métodos propios + el método sobrescrito).

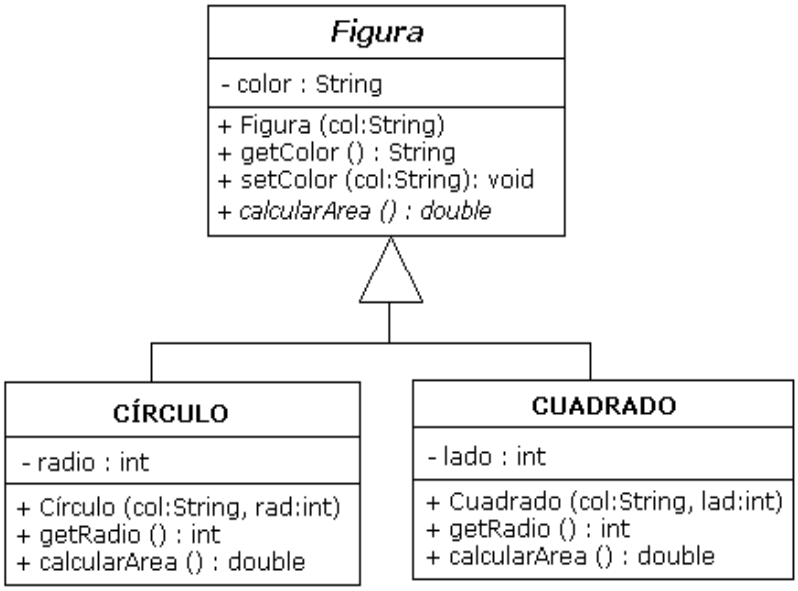


Dentro de los métodos polimórficos, existe a su vez una subcategoría de métodos, denominados **métodos abstractos**. Un método abstracto es aquel que está originalmente definido en una Clase abstracta, pero que ***no*** está implementado en dicha clase, sino exclusivamente en sus clases derivadas. Se incorporan en situaciones en las cuales se sabe que las clases derivadas deben dar un comportamiento a dicho método, pero el mismo no posee implementación a nivel de la clase base. Veremos cómo se logra esto en la etapa de implementación.

Una observación importante es que ***no*** pueden existir métodos abstractos en clases que no sean abstractas. Los métodos abstractos deben necesariamente ser implementados en las clases derivadas que no sean abstractas. En UML, los cabezales de los métodos abstractos se escriben en *cursiva* para diferenciarlos de los métodos comunes.

Ejemplo 8

Dado el siguiente fragmento de un Diagrama de Clases de Implementación:



La clase base (Figura) es *abstracta*, y además cuenta con un método abstracto que permite calcular el área de la figura. En este ejemplo, solamente existen dos tipos reales de figuras (los círculos y los cuadrados). No es posible instanciar figuras que no sean círculos ni cuadrados.

Se sabe que toda figura geométrica debe poder calcular su área, pero la forma concreta de realizar el cálculo depende del tipo concreto de figura. Nótese que el método abstracto se escribe en *cursiva* para diferenciarlo de los métodos comunes. En las clases derivadas se incluye nuevamente el cabezal método, pero esta vez en letras comunes, a efectos de representar que el mismo posee una implementación en la clase. Además, la implementación en la clase Círculo será diferente a la implementación en la clase Cuadrado. En la primera se implementará como (Pi * radio^2) mientras que en la segunda se implementará como (lado * lado).

Para finalizar esta sección, recordemos que todos los métodos polimórficos, inclusive los abstractos, siguen siendo *métodos específicos*, por lo que su incorporación al Diagrama de Clases de Implementación surgirá del *desglose de requerimientos*, como sucede con todos los métodos específicos.

Separación en Capas

En esta última sección nos ocuparemos de hablar de un concepto también muy importante en el Diseño de Sistemas Orientados a Objetos, el concepto de *Separación en capas*. Cuando iniciamos nuestro estudio de la Programación Orientada a Objetos, mencionamos algunas cualidades deseables que todo software de calidad debe poseer. Ellas eran: *mantenibilidad, extensibilidad*, *reusabilidad* y *adaptabilidad*. El principio de Separación en Capas tiene como objetivo contribuir aún más al cumplimiento de dichas cualidades.

El principio de Separación en Capas establece que todo Sistema de Software debería estar separado en al menos tres capas claramente diferenciadas. Ellas son la **Capa de Interfaz Gráfica**, la **Capa Lógica o de Negocio** y la **Capa de Persistencia** de los datos. En términos de Sistemas Orientados a Objetos, la idea es separar las Clases que manejan la interfaz gráfica por un lado, las Clases que manejan la lógica que rige la resolución de los requerimientos por otro y las Clases que manejan la persistencia de datos en disco por otro.

Un diseño Separado en Capas ayudará a, entre otras cosas, reducir la necesidad de modificar código fuente y recompilar. Por ejemplo, si en el futuro deseamos modificar la interfaz gráfica para que interactúe con los usuarios a través de ventanas o páginas web en lugar de hacerlo mediante una consola (que es como hasta ahora hemos interactuado con los usuarios), solamente modificaremos las Clases de la Capa Gráfica, ***sin tener que modificar ni recompilar ni siquiera una clase dentro de la Capa Lógica***.

En este curso estudiaremos las primeras técnicas utilizables para lograr una buena Separación en Capas. Dado que es un primer curso de Orientación a Objetos trabajaremos fundamentalmente sobre las clases de la **Capa Lógica**, dejando el estudio más detallado de las otras dos capas (Gráfica y Persistencia) para cursos posteriores. Nos ocuparemos principalmente de asegurar que la Capa Lógica **no** interactúe directamente con los usuarios (lo que compete a la Capa Gráfica) y **no** respalde ni recupere información almacenada en disco (lo que compete a la Capa Persistencia).

Por esta razón es que las Clases de la Capa Lógica ***nunca deben poseer métodos que***:

- *Carguen datos por teclado u otro mecanismo de interacción con usuarios.*
- *Muestren datos por pantalla u otro mecanismo de interacción con usuarios.*
- *Almacenen datos en archivos u otro mecanismo de persistencia.*
- *Lean datos desde archivos u otro mecanismo de persistencia.*

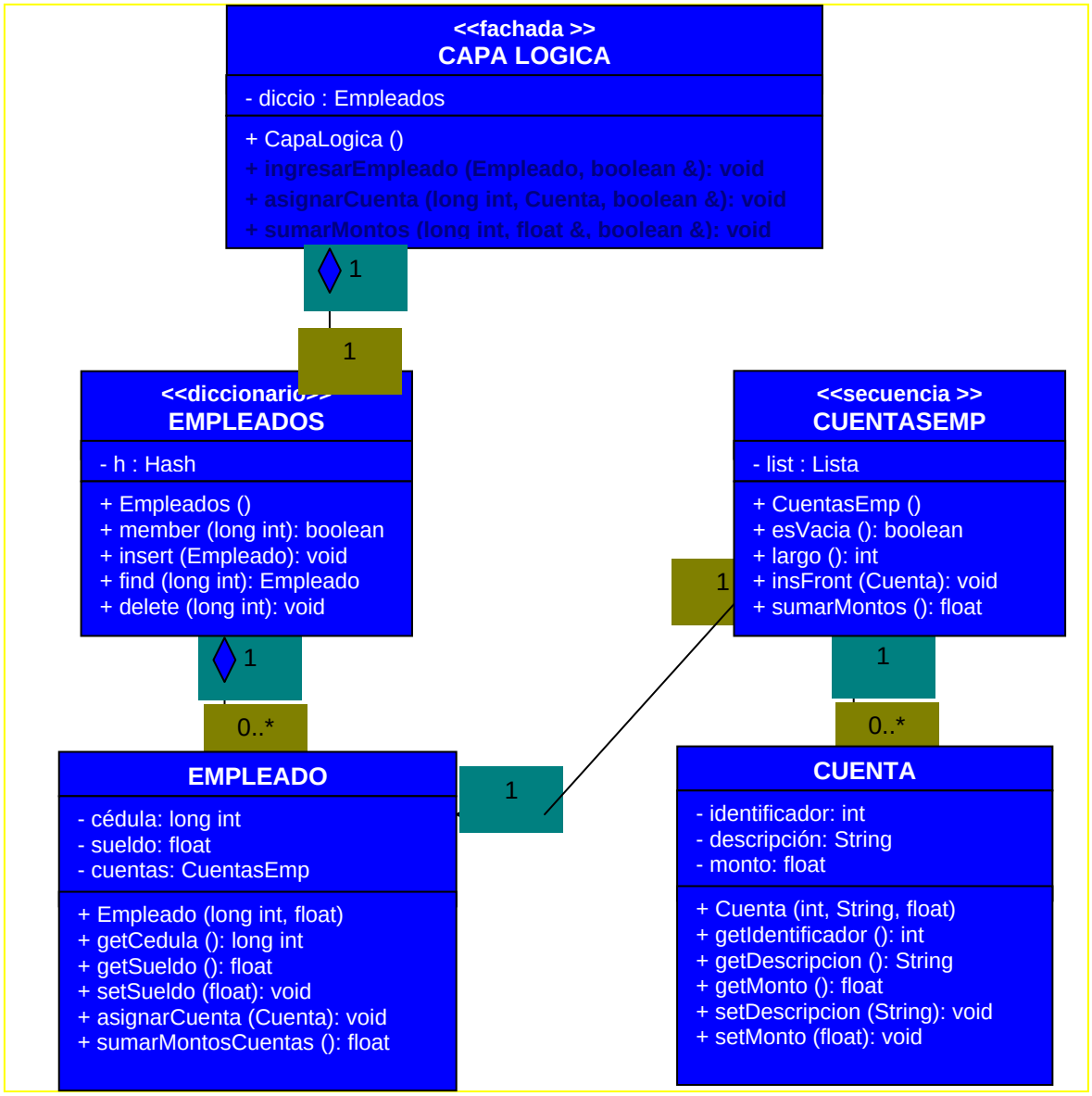
Otro paso para lograr que la Capa Lógica quede lo más desacoplada posible de las otras dos capas es proveer al Sistema de una Clase especial que sirva como ***punto único de acceso a la Capa Lógica***. Esta clase suele tener diversas denominaciones: hay quienes la llaman *Fachada*, hay quienes le dicen *Manejador* o *Handler*, o simplemente *CapaLógica*. Más allá de su nombre, lo importante es que sea el único punto de comunicación entre las distintas capas.

La Fachada será una Clase más que incorporaremos al Diagrama de Clases de Implementación. Sus métodos serán tales que permitan resolver **todos** los requerimientos del Sistema, mientras que sus atributos serán aquellas Colecciones de Objetos (definidas en la elección del diseño) que impliquen un punto de partida para la resolución de los requerimientos. La cantidad de métodos que tendrá la Fachada dependerá de la cantidad de requerimientos del Sistema. Usualmente se suele definir un método por cada requerimiento (aunque dependiendo de cada diseño puede ocurrir que sea más de uno).

La implementación de cada método de la Fachada estará directamente determinada por el *desglose* del requerimiento correspondiente. Dentro del método es donde se ejecutarán **todos** los pasos definidos en el desglose, para lo cual internamente se invocarán a aquellos métodos de las Clases (tanto primitivos como específicos) que sean necesarios para resolver cada uno de los pasos del desglose.

Ejemplo 9

Finalizaremos este capítulo incorporando una Fachada al Sistema del ejemplo 6.



Esta fachada será el único punto de acceso a la Capa Lógica. En otras clases (las de la Capa Gráfica, que no se dibujan en el diagrama) se realizará la interacción con los usuarios y la forma de ingresar datos a la Capa Lógica u obtener datos de ella para luego listarlos se hará únicamente invocando a los métodos de la fachada. Similar forma de interacción existirá entre la Capa Lógica y la Capa de Persistencia.

La fachada posee como **único** atributo al Diccionario de Empleados, no posee ninguna Secuencia de Cuentas como atributo. Recordar que, de acuerdo al diseño elegido, en realidad existen muchas instancias de la Clase **CuentasEmp** (una por cada empleado), cada una de las cuales es un atributo del Empleado al que pertenece. Además, de acuerdo con el *desglose de requerimientos*, cada uno de ellos inicia su ejecución a través del Diccionario de Empleados.

Analizamos ahora los métodos de la Fachada. Obsérvese que se incorporó un método por cada requerimiento definido. Vamos a comparar cada requerimiento con el cabezal que le corresponde:

Requerimiento 1: Ingresar un nuevo empleado.
Cabezal correspondiente: **ingresarEmpleado (Empleado, boolean &): void**

De acuerdo con el desglose de este requerimiento, los datos de entrada eran los del nuevo empleado, por esta razón le pasamos como parámetro el objeto empleado a almacenar. Nótese que, de acuerdo con la Separación en Capas, los datos de dicho empleado fueron obtenidos del usuario en forma previa a la invocación de este método. Como parámetro de salida, se devuelve un booleano que indica si el requerimiento fue exitoso o no. Esto responde nuevamente al principio de Separación en Capas. Dentro de la Capa Lógica no corresponde emitir mensajes de error al usuario, por lo que se devuelve un indicador del resultado para que luego la Capa Gráfica sepa si debe o no desplegar un mensaje de error.

Requerimiento 2: Dada la cédula de un empleado, asignarle una nueva cuenta. Su identificador será un número correlativo al último número de cuenta manejada por dicho empleado.
Cabezal correspondiente: **asignarCuenta (long int, Cuenta, boolean &): void**

Para este cabezal valen las mismas consideraciones que para el requerimiento 1. Nótese que además de la nueva cuenta, como dato de entrada se ingresó también la cédula del empleado a buscar, de acuerdo con el desglose del requerimiento 2.

Requerimiento 3: Dada la cédula de un empleado, sumar los montos de todas sus cuentas.
Cabezal correspondiente: **sumarMontos (long int, float &, boolean &): void**

En ese caso el único dato de entrada, de acuerdo con el desglose, es la cédula del empleado. Hay dos parámetros de salida: el valor de la suma de montos y nuevamente el booleano que indica si el requerimiento fue exitoso o no.

Obsérvese que en este ejemplo, los tres métodos de la fachada son **procedimientos**. Esto es así en este diseño en concreto, porque los requerimientos así lo exigieron. Puede haber métodos de fachadas que sean funciones, si es que los requerimientos así lo exigen. De acuerdo con los criterios usuales para elegir entre función o procedimiento, optaremos por incorporar funciones *únicamente* cuando interese devolver **un** único resultado y **no** se modifiquen datos internos a la Capa Lógica.

En este ejemplo, el manejo de errores se hizo devolviendo un valor booleano. Se hizo así porque el único error posible en los tres requerimientos propuestos tiene que ver con la existencia o no del empleado en cuestión. Pero podrían existir otras situaciones en las cuales haya dos o más tipos de error posibles para un mismo requerimiento. En tal caso, no existe una estrategia universal para indicar la ocurrencia del error. Hay quienes prefieren devolver un valor booleano por cada tipo de error posible, o bien devolver un enumerado con tantos valores posibles como tipos de error existan.

La elección de la estrategia para el manejo de errores depende de cada diseño en concreto. En cualquier caso, siempre se debe garantizar que los posibles mensajes de error **nunca** serán directamente emitidos al usuario desde el interior de la Capa Lógica.