Programación con TADs
Programación Orientada a Objetos
(apuntes del profesor)

Juan Antonio Romero del Castillo (aromero@uco.es)
Departamento de Informática y Análisis Numérico
Universidad de Córdoba

16 de octubre de 2018

Índice general

Capítulo 2

Abstracción y especificación: el camino hacia la orientación a objetos

2.1. Introducción

Parte del apartado 1.26 de [DD03].

- La orientación a objetos es una manera natural de pensar en la vida real y en la labor de programar.
- Entonces, ¿por qué no empezamos programando con objetos?. La respuesta es: por simplicidad de aprendizaje, puesto que aún usando objetos, éstos se componen de piezas de programas estructurados. Por ello es conveniente conocer antes los principios básicos de la programación estructurada.

2.1.1. Objetos del mundo real

Algunos aspectos del mundo real que se usan de forma natural en la programación orientada a objetos:

- En el mundo real hay objetos/entidades: personas, animales, libros... Es habitual pensar en términos de objetos.
- Son abstracciones porque las personas tenemos la habilidad de "abstraernos" de muchos detalles cuando creamos abstracciones para simplificar y comprender mejor las cosas. Por ejemplo, pensar en términos de playa en vez de granos de arena, de bosques en vez de en árboles uno junto al otro, de casas en lugar de ladrillo sobre ladrillo, etc.
- Todos los objetos del mundo real tienen un comportamiento. Y dichos objetos comprenden mejor interactuando con ellos: conversando con ellos, usándolos, mirándolos, etc.

- Todos tienen **atributos**: tamaño, forma, color, etc.
- Todos presentan un **comportamiento** que los definen u operaciones que realizan: la pelota bota, el coche acelera, etc.
- Así, vemos que un objeto se entiende mejor estudiando sus atributos y su comportamiento.
- Otro aspecto importante de estos objetos del mundo real es que se relacionan unos con otros: se parecen entre si, unos se hacen utilizando otros, se comportan parecido, comparten atributos, etc.

2.1.2. La POO

Aplicar esta herramienta tan interesante que es la abstracción en la labor de programar es una de las ideas básicas del paradigma de la programación orientación a objetos.

Dicho paradigma centra la labor de programar en los objetos y sus distintas clases que nos podemos encontrar en nuestros programas, la relación entre ellos, su comportamiento, etc.

Algunas consideraciones:

- Es una forma más natural y cercana a la realidad de programar.
- La base de la programación estructurada es la función, que es más dificil de comprender.
- La POO usa como base la **clase** que integra los siguientes conceptos:
  - Representa un objeto del mundo real.
  - Tiene atributos.
  - Tiene comportamiento.
  - Tiene entidad por si misma.
  - Tiene sentido por si misma, se entiende, se comprende fácilmente.
  - Se puede, por tanto, reutilizar en distintos problemas.
  - Se pueden establecer de forma natural relaciones de distintos tipos entre objetos, etc.

La POO es una forma natural de modelar problemas reales mediante programación.

Existen criterios para evaluar si un método, lenguaje o herramienta son orientados a objetos o no. El capítulo 2 de [Mey99] describe los elementos fundamentales que todo LPOO debe tener (es una lectura recomendada al alumno). En dicho capítulo se describen uno a uno dichos elementos: clases, aserciones, ocultación de la información, excepciones, genericidad, herencia, polimorfismo, ligadura dinámica, etc. que nosotros iremos desarrollando a lo largo de la asignatura.

2.2. Descomposición y Abstracción.

Una de las tareas más importante en el desarrollo de software es la descomposición. La descomposición de un problema en módulos (la modularidad o modularización) es algo que se utiliza desde los comienzos de la programación. Pero no todos los métodos de descomposición modular son buenos.

En este apartado veremos como la abstracción de datos (los TADs) implica un método natural y de calidad de descomposición modular. Pero para llegar ahí, primero tenemos que saber **qué es un buen método de descomposición modular**.

2.2.1. Criterios, reglas y principios de descomposición modular

Del capítulo 3 de [Mey99] podemos extraer estos criterios, reglas y principios de descomposición modular. La idea aquí es mostrar como la POO además de las ventajas que hemos visto que presenta, es también una buena técnica de descomposición modular. El alumno debe leer esta sección completa y analizar cada uno de estos criterios, reglas y principios de descomposición modular.

- Se puede resaltar la importancia de la descomposición en la labor del programador diciendo que precisamente la tarea de un programador es esa: realizar una descomposición de calidad.
- La calidad de una descomposición modular, al igual que la calidad del software, no es un algo subjetivo o difuso. El propósito de esta sección es refinar y formalizar la definición de este concepto.
- Haremos referencia al ejemplo del tema anterior del videoclub en los siguientes apartados dejando ver que la POO hace una descomposición modular buena, mientras que con la TDD cuesta más trabajo.

A partir de un problema desarrollamos una solución software utilizando un método de desarrollo. Ese método concreto pertenecerá a una metodología de desarrollo que será el cuerpo teórico que la defina.

El software resultante será un sistema formado normalmente por varios módulos que se interelacionan entre sí. Los módulos y su forma de relacionarse se denomina **arquitectura del sistema**.

El diseño de los módulos que compondrán la arquitectura del sistema es una labor muy importante.

2.2.2. Criterios/Requisitos para una buena descomposición modular

Un método de descomposición modular para ser bueno debe satisfacer los siguientes 5 requisitos fundamentales:

1. Descomposición modular.
   - **Def.** Un método de construcción de software debe ayudar en la tarea de descomponer el problema de software en un pequeño número de subproblemas menos complejos, interconectados mediante una estructura sencilla, y suficientemente independientes para permitir que el trabajo futuro pueda proseguir por separado en cada uno de ellos.
   - **Comentarios.** Existen tecnologías que de forma natural favorecen el proceso general de descomposición de un problema. La POO, las tecnologías que usan patrones de diseño, etc. Otras tecnologías, no.
2. Composición modular.
   - **Def.** Un método satisface la composición modular si favorece la producción de elementos software que se puedan combinar libremente unos con otros para producir nuevos sistemas, posiblemente en un entorno bastante diferente de aquel en que fueron desarrollados inicialmente.
   - **Comentarios.** Los módulos son independientes y fáciles de combinar entre si para varias aplicaciones. Bibliotecas de programas, comando pipe/pipeline (|) de la Shell de UNIX, etc. Como contraejemplo los módulos que dependen de la aplicación en la que se ha desarrollado y no se pueden usar en otras.
3. Comprensibilidad modular.
   - **Def.** Un método favorece la Comprensibilidad Modular si ayuda a producir software en el cual un lector Humano puede entender cada módulo sin tener que conocer los otros, o, en el peor caso, teniendo que examinar sólo unos pocos de los restantes módulos.
   - **Comentarios.** Afecta mucho a la compresión y al mantenimiento. Se suele asumir que si nos resulta dificil explicar qué hace o para qué se ha creado un módulo, ese módulo no está bien diseñado (lo mismo ocurre a otros niveles con los procedimientos o funciones). Los módulos demasiado grandes tampoco ayudan. Además, los módulos demasiado grandes son poco reutilizables.
4. Continuidad modular.
   - **Def.** Un método satisface la Continuidad Modular si en el sistema resultante desarrollado con ese método, un pequeño cambio en la especificación o en los requisitos provoca sólo cambios en un único módulo o en un número reducido de ellos.

- **Comentarios.** El mantenimiento es una fase muy importante del ciclo de vida del software. Si no nos preparamos para él, fracasaremos. Si nuestros módulos no facilitan el mantenimiento, no serán unos buenos módulos ya que el cambio en cualquier fase del desarrollo y durante el propio mantenimiento es algo bastante frecuente.

5. Protección modular.

   - **Def.** Un método satisface la Protección Modular si produce arquitecturas en las cuales el efecto de una situación anormal que se produzca dentro de un módulo durante la ejecución queda confinado a dicho módulo o en el peor caso se propaga sólo a unos pocos módulos vecinos.
   - **Comentarios.** Un error dentro de un módulo debe ser tratado en dicho módulo y solucionado allí si es posible. Si un error dentro de un módulo no es detactado por el módulo y siguen ejecutandose otros módulos arrastrando el error, éste será dificil de corregir y de impredecibles consecuencias.

2.2.3. Reglas generales para una buena descomposición modular

Cuando vayamos a hacer una descomposición, para que el resultado sea de calidad, podemos guiarnos de estas reglas. Cada regla afecta a uno o a varios de los criterios anteriores. Estas reglas deben seguirse para asegurarnos cumplir los requisitos anteriores.

1. Correspondencia directa.

   - **Def.** La estructura modular obtenida debe ser compatible con la estructura modular del dominio del problema.
   - **Comentarios**. Si ya se ha estructurado el problema en módulos, se han identificado, etc., su correspondencia con módulos software ayudará mucho en el desarrollo del sistema.

2. Pocas interfaces.

   - **Def.** Un módulo debe comunicarse con el menor número posible de módulos.
   - **Comentarios**. Cada módulo debe intentar ser lo más independiente posible, eso ayuda en genaral a la buena descomposición. Si un módulo se comunica o interactúa con muchos módulos es que depende de ellos en gran medida y esto, en general, no es lo ideal.

3. Pequeñas interfaces.

   - **Def.** Los módulos deben intercambiar la menor información posible.

- **Comentarios**. "Pequeñas interfaces"tendría la misma argumentación que el apartado anterior "pocas interfaces".

4. Interfaces explícitas.
   - **Def.** Las interfaces deben ser obvias a partir de su simple lectura.
   - **Comentarios**. La claridad del módulo (la autodocumentación que veremos luego como principio), afecta en gran medida a su calidad y, por tanto, es determinante para que un módulo sea bueno. Lo mismo ocurre si un módulo debe llamar a otro, debe entenderse de forma lógica el motivo de esa llamada.
5. Ocultación de la información.
   - **Def.** El diseñador de cada módulo debe seleccionar un subconjunto de propiedades del módulo como información oficial sobre el módulo para ponerla a disposición de los autores de módulos clientes.
   - **Comentarios**. El resto de información será privada (encapsulada, oculta) y no debe hacerse pública puesto que no es necesario que se conozca para el uso del módulo, es más, su conocimiento puede resultar contraproducente puesto que será información de bajo nivel de la cual el usuario no debe depender.

2.2.4. Principios para una descomposición modular de calidad

A partir de las reglas y criterios anteriores de pueden derivar algunos principios a seguir para una descomposición modular de calidad. Estos principios refuerzan, repiten y concretan en algunos casos lo ya dicho.

1. Unidades modulares lingüísticas. **Def.** Los módulos deben corresponderse con las unidades sintácticas del lenguaje de programación que se utilice. Por ejemplo, si hacemos un módulo necesitamos un lenguaje que tenga un buen soporte de módulos. Que se puedan compilar por separado, etc. De otra forma habría que hacer **traducciones** indeseables, adaptaciones complejas y costosas. El esfuerzo y la complejidad del diseño sería mayor y mayor por tanto la posibilidad de error.

2. Auto-documentación. **Def.** El diseñador de un módulo debiera esforzarse por lograr que toda la información relativa al módulo forme parte del propio módulo. El ideal sería que la simple lectura del código del módulo es su propia documentación.

3. Acceso uniforme. **Def.** Todos los servicios ofrecidos por un módulo deben estar disponibles a través de una notación uniforme sin importar si están implementados a través de almacenamiento o de un cálculo. La uniformidad del sistema es importante. Ya que el usuario hace un esfuerzo por entender el módulo, no le cambiemos la nomenclatura, las reglas básicas, etc. en el otro módulo.

4. Principio Abierto-cerrado. Puede parecer contradictorio pero son elementos de naturaleza diferente.
   - **Def.** Los módulos deben ser a la vez abiertos y cerrados.
   - Abierto: esforzarse en diseñar el módulo de manera que quede abierto a facilitar la posterior modificación, ampliación, etc. (considerar, como hemos dicho antes, que el 70 % del coste del software se dedica al mantenimiento). Para ello se diseñan arquitecturas abiertas, se siguen estándares y convenciones, etc.
   - Cerrado: El módulo estará cerrado si está disponible para ser usado e integrado en el sistema. Su diseño abierto no impedirá que el módulo quede completado y listo para ser usado en la aplicación para la que se ha creado.
5. Elección única. Relacionado con la ocultación de la información. Muchas veces haremos un software que, por ejemplo, usa una estructura de datos susceptible de ser ampliada o modificada en un futuro. Para que esa elección entre una estructura de datos y otra no afecte al resto de módulos, solo un módulo debe acceder a dicha estructura de datos. Y este módulo será el único que se modifique al ampliar/cambiar la estructura de datos. Este módulo será el único que elija (de ahí la elección única) opciones según la estrutuaras de datos utilizada en cada momento. En el resto de módulos no influirán esta elección porque para ellos estarán ocultas.

2.2.5. Conclusiones

- La adecuada descomposición, la modularidad, de un proyecto software es fundamental.
- No todas las descomosiciones modulares satisfacen los criterios, reglas y principios de una buena descomposición modular. Hay que tener cuidado en ello y hacer buenos diseños.
- Existen tecnologías de programación que no favorecen la buena descomosición.
- POO y la abstracción de datos en general, es un método de descomposición que satisface todo lo dicho en esta sección.

2.3. Abstracción

Estamos hablando de abstracción desde el comienzo del curso como el concepto que permite el desarrollo de la orientación a objetos. Veamos en esta sección algunas definiciones de este concepto y de conceptos relacionados que nos permiten poner en marcha los Tipos Abstractos de Datos (TADs) en nuestros programas.

Podemos definir abstracción como un proceso mental que tiene dos aspectos complementarios (gran parte de lo que sigue es de [LG01]):

1. **Destacar** los detalles importantes (relevantes) del objeto de estudio.
2. **Ignorar** los detalles irrelevantes del objeto de estudio.

La abstracción permite estudiar fenómenos complejos utilizando un método jerárquico con sucesivos niveles de detalle.

El uso de abstracciones en informática está muy extendido en diferentes campos. Por ejemplo en sistemas operativos el simple concepto de fichero es una abstracción que facilita el uso del disco, y hay otras muchas abstracciones de este tipo tanto en sistemas operativos, como en bases de datos, como en lenguajes de programación, etc.

2.4. Abstracción por parametrización

- Es el uso de parámetros propiamente ya supone el uso de una abstracción importante ya que el parámetro permite abstraer de los detalles del dato sobre el que se aplica el procedimiento o función.
- Un procedimiento que se realiza sobre un parámetro A, permite que luego A sea sustituido en la invocación por cualquier dato y en algunos casos incluso de diferente tipo.

2.5. Abstracción por especificación

- La especificación permite abstraer de los detalles de el **cómo**, considerando únicamente el **qué**.
- Nos informa de forma precisa de los detalles **relevantes** de datos, funciones, etc. y nos evita los detalles **irrelevantes**.

2.5.1. Tipos de especificaciones

1. **Formales**. Usando el lenguaje matemático y el de la Lógica. Son exactas y precisas como el lenguaje de la matemática y la lógica.
2. **Informales**. Usando el lenguaje natural. Son imprecisas, como el lenguaje natural. Nosotros utilizaremos este tipo, pues son sencillas y también muy útiles. Aunque nos esforzaremos para que sean lo más claras, exactas y precisas que podamos usando lenguaje natural.

Veremos dos tipos: de procedimientos, y de datos (TADs).

2.6. Abstracción de procedimientos o abstracciones procedimentales

- Irrelevante: El cómo, y en este caso es: **la implementación**.
- Relevante: El qué: **la especificación**.

2.6.1. Especificación de procedimientos

Usaremos la siguiente plantilla que deberá ir a modo de comentario delante de cada procedimiento (función en C/C++) en nuestro código fuente.

```
PROC nombre_proc(lista y tipos de los parámetros)
     DEV (tipo devuelto)

REQUIERE ........

MODIFICA ........

EFECTOS  .......
```

La cláusula **REQUIERE** se utiliza para describir las condiciones que deben cumplirse antes de la ejecución del procedimiento. Si existen condiciones se dice que el procedimiento es *parcial* (esto no es deseable ya que produce procedimientos no robustos). Si no existen se utiliza el símbolo *true* (quiere decir que la cláusula REQUIERE siempre es verdadera y se cumple siempre) y el procedimiento es *total*.

La cláusula **MODIFICA** es una enumeración de los parámetros, de la lista de parámetros, que se modificarán al ejecutarse el procedimiento. No se explica como será esa modificación, simplemente se enumeran. Si el procedimiento no modifica ningún parámetro se se deja vacío utiliza el símbolo $\emptyset$.

La cláusula **EFECTOS** es un texto en lenguaje natural (posiblemente extendido con notación matemática) mediante el cual se explica de forma muy clara y precisa lo que hace el procedimiento haciendo referencia explícita a cada parámetro, a si se modifican o no, y al valor devuelto si lo hubiera. Si no se cumple la condicion o condiciones de la cláusula REQUIERE, lo que diga esta cláusula no tiene ningun valor y los efectos del procedimiento podrán ser impredecibles.

2.6.2. Ejemplos: concatenar, borrarDuplicados, busquedaBinaria

PROC concatenar(string a, string b) **DEV** (string ab)
REQUIERE True
MODIFICA $\emptyset$

EFECTOS al finalizar, “ab” es una cadena que contiene los caracteres de “a”, en el mismo orden que en “a”, seguidos de los caracteres de “b”, en el mismo orden que en “b”.

PROC borrarDuplicados(entero a[]) **DEV** ()
REQUIERE True
MODIFICA a
EFECTOS borra todos los elementos duplicados de “a”dejando el primero intacto. Por ejemplo, si a=[3,13,3,6] antes de la llamada, el resultado de la llamada será a=[3,13,6].

PROC busquedaBinaria(entero a[], entero x, entero n) **DEV** (entero i)
REQUIERE “a” esté ordenado ascendentemente
MODIFICA ∅
EFECTOS Si “x” está en “a”, “i” es tal que a[i]=x; en otro caso “i” es una unidad mayor que el tamaño del vector, es decir, i=n+1.

2.7. Excepciones

Las excepciones son un mecanismo cada vez más utilizado en los lenguajes de programación actuales para notificar situaciones de error o situaciones excepcionales (de ahí su nombre) al usuario.

Un procedimiento parcial produce programas no robustos, ya que al imponer condiciones previas a los parámetros pueden no cumplirse siempre y dar lugar a errores. Los procedimientos totales, por tanto, mejoran la robustez de los programas.

Pero si hacemos un procedimiento total y detectamos algún problema con los parámetros (o con otras situaciones) ¿cómo notificar que hay un problema?.

Normalmente lo mejor es devolviéndole un código de error. En casi todos los casos esto es lo habitual y lo que debe hacerse porque es suficiente. Aunque hay casos en que esto no es lo mejor.

Comprobar errores y devolver un código de error puede tener inconvenientes en algunos casos:

- El usuario se tiene que tomar la molestia de comprobar el valor devuelto.
- Si el valor devuelto en caso de error es del mismo tipo que en caso de no-error, ¿cómo distinguirlos?. Por ejemplo, cuando se quiere devolver uno de los valores de un vector de enteros y la posición pasada como parámetro no existe.
- Cuando se busca que una función sea muy muy rápida y se requiere a toda costa que su tiempo de ejecución sea mínimo, entonces incluso nos planteamos quitar

las comprobaciones de error porque dichas comprobaciones hacen la función más lenta. ¡Ojo!, esto ocurre solo en contadas ocasiones; evitar comprobaciones solo debe hacerse en este caso y siempre de forma justificada.

Para estos casos fundamentalmente existe un mecanismo denominado "excepciones" que permite manejar estas situaciones excepcionales.

2.7.1. Especificación de excepciones

PROC busquedaBinaria(entero a[], entero x) **DEV** (entero i) **EXCEPCIONES** ExcepcionNoEncontrado
REQUIERE "a" esté ordenado ascendentemente
MODIFICA ∅
EFECTOS Si "x" no está en "a" se lanza la excepción *ExcepcionNoEncontrado*; Si "x" está en "a", "i" es tal que a[i]=x.

PROC factorial(entero n) **DEV** (entero i) **EXCEPCIONES** ExcepcionNoPositivo
REQUIERE True
MODIFICA ∅
EFECTOS Si "n" no es positivo lanza la excepción *ExcepcionNoPositivo*; sino "i" es el factorial de "n".

2.7.2. try, catch

Para ver como se usan las excepciones en un LPOO veamos el caso de C++, por ejemplo. Cuando un procedimiento puede lanzar excepciones se puede poner dentro de un bloque try. Y cada una de las excepciones que lance se pueden procesar en bloques catch.

Si el procedimiento `proc1(entero x, entero y)` puede lanzar la excepción `A` y la excepción `B` entonces podría escribirse el siguiente código:

```
try {
   proc1(i,j)
    }
catch (A)
    {
    ........
    }
catch (B)
    {
    ........
```

```
}
```

El código dentro de cada bloque se ejecutaría en el caso de que se diera alguna de las excepciones A, o B respectivamente. En otros LPOO la sintaxis es similar.

2.8. Abstracción de datos

- Consiste en la especificación de TADs.
- Definición de TAD: *Es una colección de datos y operaciones sobre estos datos, que se define mediante una especificación que es independiente de cualquier implementación.*

2.8.1. Especificación de TADs

Utilizaremos la siguiente plantilla que deberá ir a modo de comentario delante de cada clase en nuestro código fuente.

```
TAD nombre_TAD

DESCRIPCIÓN
           ..................
OPERACIONES


PROC operación1 DEV (...)
......

PROC operación2 DEV (...)
......
......
```

- El concepto de **interfaz** de un TAD (la parte pública de una clase).
- El concepto de **implementación** de un TAD (la parte privada de una clase).
- El concepto de **encapsulamiento de la información**.
- Clase: características y atributos; comportamiento y operaciones (métodos).

2.8.2. Ejemplo: el TAD Fecha

Mediante estructuras de datos nada evitaría cometer errores como los siguientes u otros muchos:

```
fechaNacim.d=64;
fechaNacim.m=36
fechaNacim.a=-28
```

El TAD permite dotar al programador con todo el control, con lo que el resultado será de más calidad.

Este TAD deberá completarse e implementarse en C++ como trabajo complementario del alumno.

TAD Fecha
DESCRIPCIÓN
El TAD Fecha representa y gestiona una fecha válida del calendario.
OPERACIONES
PROC set(entero nd, entero nm, entero na) **DEV** (entero)
REQUIERE True
MODIFICA $\emptyset$
EFECTOS asigna a la fecha el dia "nd", el mes "nm", y el año "na", comprobando previamente que sea una fecha válida, en cuyo caso devuelve OK. Si la fecha pasada como parámetro no es válida la fecha actual no se modifica y se devuelve uno de los siguientes errores:

- ERROR_A si el año es erróneo.

- ERROR_M si el mes es erróneo.

- ERROR_D si el día es erróneo.

PROC getd() **DEV** (entero)
REQUIERE True
MODIFICA $\emptyset$
EFECTOS devuelve el día de la fecha. (análogo para: *getm()* y *geta()*)

PROC setd(entero nd) **DEV** (entero)
REQUIERE True
MODIFICA $\emptyset$
EFECTOS Si la nueva fecha formada por el día "nd", y el mes y el año almacenados es válida, se llevará a cabo la modificación del día al valor "nd". En caso contrario se devuelve el valor ERROR_D. (análogos para: *setm()* y *seta()*)

Comentarios

- Las funciones de la clase Fecha no tendrán dentro E/S a pantalla o teclado. Toda interacción con ellas es mediante parámetros y valores devueltos.
- El nombre de cada función, de los errores, y de cada identificación en general debe escogerse con buen criterio: comprensibilidad, uniformidad, etc.
- Todas las funciones deben tener control de errores para que nunca pueda darse el caso de que el usuario introduzca en un objeto de tipo Fecha una fecha errónea.

2.8.3. Idea básica: Los TADs son independientes de la implementación.

Las ventajas directas son:

- Mantenimiento: no afectan al usuario los cambios internos, futuras versiones, mejoras de eficiencia, etc.
- Facilidad de uso: el acceso al interior complica la labor del usuario. Además, el usuario ahorra código al manipular el interior con la interfaz.
- Uso más coherente: no hay posibilidad de mal uso, errores, etc., por parte del usuario.
- Suelen ser más Reusables, ya que no hay dependencias de bajo nivel.
- En general se obtiene un TAD de más calidad.

2.9. La programación con TADs

- La decisión de implementación se toma al final. Veamos un ejemplo en el que resolvemos un problema, definimos un TAD y lo especificamos sin referirnos nunca a su implementación.
- La decisión de implementación se toma al final

2.9.1. El Fichero estadístico. Enunciado del problema

Diseñar un programa que lee enteros de un fichero y muestra, por cada entero distinto leído, una estadística mostrando su frecuencia de aparición en el fichero.

2.9.2. El Fichero estadístico. Solución con TADs

Lo haremos en pseudocódigo para resaltar la independencia del Lenguaje de Programación.

Este ejercicio ilustra la forma de programar usando TADs. Antes de empezar a darle solución no tengo ningun TAD, incluso escribo el siguiente algoritmo antes de tener los TADs definidos, pero el propio algoritmo me irá diciendo cómo serán los TADs que necesito para resolverlo.

Para resolver este problema supongo la existencia del TAD Tabla (o como quiera llamarlo) que me gestionará los enteros.

```
Algoritmo estadistica(fichero f)
Inicio
  mientras no fin(f) hacer
    leer(f,x)
    añadir(t,x) // o bien t.insert(x)
  finmientras
  para i de 1 a total(t) hacer
    x=infonum(t,i) // o bien x=t.infonum(i)
    frecx=infofrec(t,x)  // o bien frecx=t.infofrec(x)
    escribir("entero ",x, "frecuencia =", frecx)
  finpara
Fin
```

Quedan aplazados los detalles de implementación del del TAD Tabla.

2.9.3. El Fichero estadístico. El TAD Tabla

TAD Tabla
DESCRIPCIÓN
Gestiona un conjunto de enteros y sus estadísticas.
OPERACIONES
PROC añadir(entero i) **DEV** ()
REQUIERE True
MODIFICA $\emptyset$
EFECTOS incrementa la frecuencia del entero i en una unidad.

PROC total( ) **DEV** (entero n)
REQUIERE True
MODIFICA $\emptyset$
EFECTOS devuelve en "n" el número de enteros distintos en la tabla.

PROC infonum(entero i) **DEV** (entero x) **EXCEPCIONES** FUERA_DE_RANGO

REQUIERE True
MODIFICA ∅
EFECTOS devuelve el entero "x" que ocupa la posición i-ésima. Si la posición "i" no existe, se lanza la excepción: FUERA_DE_RANGO

PROC infofrec(entero x) **DEV** (entero f)
REQUIERE True
MODIFICA ∅
EFECTOS devuelve en "f" el número de veces que se repite "x" en la tabla.

2.10. Implementación de TADs

- Significado de "cumplir una especificación dada".
- Minimizar el cumplimiento de la especificación: si nos piden la ocurrencia de un elemento en un vector, devolver: si o no; no su posición ni su ocurrencia.
- Generalización: más generalización, menos eficiencia.
- Siempre simplicidad. Cuanto más simple sea una función mejor. Dicen que si cuesta trabajo pensar un nombre para una función, suele ser porque la función está mal pensada.

2.10.1. STUBS: una ayuda a la programación con TADs

Algunos programadores usan STUBS para el pronto desarrollo de un prototipo de sus programas. Cuando una función o un método aún no está codificado se construye una breve versión preliminar que permite que dicha función o método pueda llamarse aunque lo que haga no sea lo que hará su versión final.

Un procedimiento STUB puede contener, por ejemplo, un mensaje de "estoy aquí" indicando que se está ejecutando, su nombre y el valor de los parametros que recibe. O devolver un valor razonable, o incluso pedir al usuario un valor para devolverlo y poder trabajar así con distintos valores devueltos.

Ejemplo de STUB:

```
int media(int a[], int n)
{
int valor_devuelto;

cout << ‘‘Función media() llamada con n=’’<< n << ‘‘y los valores: \n’’;
for (int i=0;i<n;i++) cout << a[i] << ’\t’;

cout << ‘‘Introduce el valor a devolver’’;
cin >> valor_devuelto;
```

```
return valor_devuelto;
}
```

2.11. Operaciones con TADs

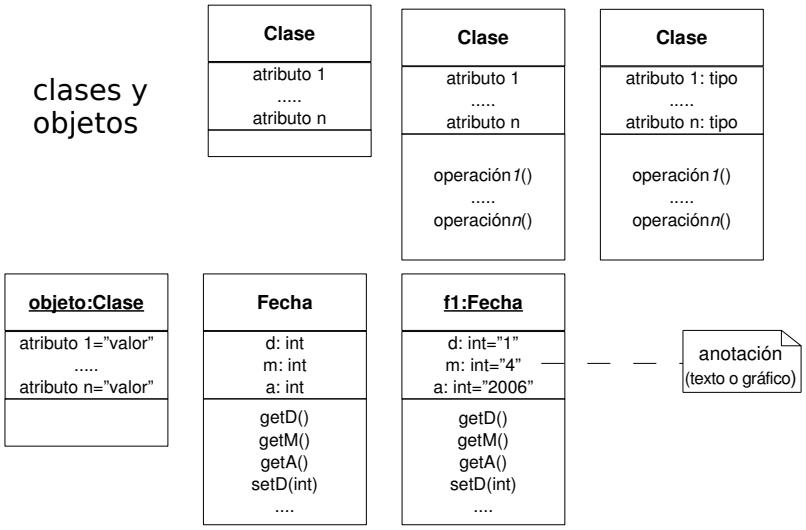
- Constructores: llevan al objeto a su estado inicial. Inicialización del objeto y sus datos internos.
- Observadores: acceso de lectura a una característica del objeto. Devuelve información del objeto sin modificarlo.
- Modificadores: acceso de escritura a una característica del objeto. Esta operación modifica de algún modo el objeto.
- Destructores: llevan al objeto a su estado final descartando posibles efectos laterales imprevistos (liberación de memoria, cierre de archivos, posibles operaciones finales que hay que realizar antes de terminar, etc.)

2.12. Un poco de UML

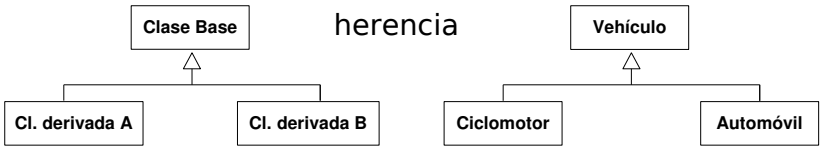
UML es un lenguaje de modelado universal que dispone de una notación ampliamente usada en la industria en general y en la documentación de software en particular.

Veremos solo algunos elementos UML (muy pocos) referentes a clases y objetos, herencia, agregación y paso de mensajes entre objetos (en la POO es común indicar gráficamente la interacción entre objetos mediante el paso de mensajes entre ellos).

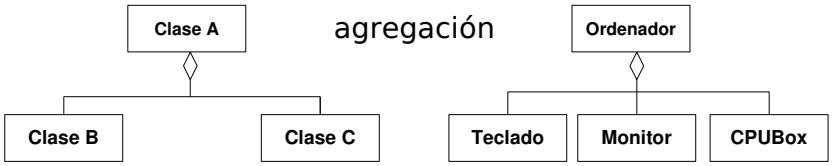
clases y objetos



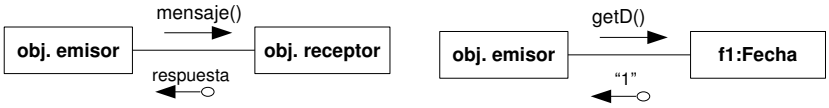
herencia



agregación



paso de mensajes



Bibliografía

[DD03] Harvey M. Deitel and Paul J. Deitel. *Cómo Programar en C++*. Pearson Educación, México, 4 edition, 2003.

[GHJV03] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Patrones de diseño*. Pearson Educación, S.A., Núñez de Balboa 120, Madrid, 2003.

[LG01] Barbara Liskov and John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, USA, 1 edition, 2001.

[Mey99] Bertrand Meyer. *Construcción de Software Orientado a Objetos*. Prentice Hall Iberia, S.R.L., Madrid, 2 edition, 1999.

[Sch04] Herbert Schildt. *C++. A Beginner's Guide. Second Edition*. McGraw-Hill/Osborne, Emeryville, California 94608, USA, 2004.