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 1

Introducción

1.1. Estructuras de Datos y Tipos Abstractos de Datos (Abstract Data Type - ADT)

1.1.1. La programación con Estructuras de Datos

- En este curso vamos a aprender el paradigma de la orientación a objetos en el desarrollo de software. En el curso pasado realizamos algoritmos que manipulan estructuras de datos, y también se aprendieron técnicas de programación estructurada, como la descomposición sucesiva de un programa en módulos agrupados en uno o varios ficheros de código fuente. Pero aunque estas técnicas pueden dar lugar a programas de calidad, la orientación a objetos es un paso adicional que ha permitido a desarrolladores de todo el mundo disponer de más herramiemtas para obtener mejores resultados en sus desarrollos. Mejores resultados en el sentido de la calidad del software desarrollado.

- Definición de Estructura de datos: *es un método de almacenamiento de los datos* en la memoria del ordenador. Una forma de estructurar datos para su almacenamiento y posterior acceso y manipulación (y sólo eso).

- Nosotros estudiaremos en este curso la programación orientada a objetos desarrollando el concepto de Tipo Abstracto de Datos (TAD), el cual es mucho más amplio y potente que el de estructura de datos y cambiará nuestra forma de programar dotándola de una mayor potencia de desarrollo y una mayor calidad del producto final (el software). Y este es el objetivo principal de este curso: mejorar sensiblemente el nivel de nuestra forma de programar y con ello la calidad de nuestros programas.

- El término "abstracción" hace referencia a su significado natural: *formar mediante una operación intelectual una idea mental o noción de un objeto extrayendo de los objetos reales particulares los rasgos esenciales, comunes a todos ellos*. Lo desarrollaremos más y lo utilizaremos con frecuencia a lo largo de este curso.

- Usar TADs aporta muchas mejoras a nuestros programas. Los programas que no usan TADs se caracterizan muchas veces por ser excesivamente complejos, difíciles de entender, crípticos y oscuros, lo cual no beneficia en nada al programa, más bien al contrario, como veremos más adelante. Veamos como ejemplo el código fuente en lenguaje C de un programa que simule el lanzamiento de dos dados:

```
// juego.c
// generación de 2 números aleatorios

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
int dado1, dado2, total, r1, r2;

srand(semilla); /* establece la semilla del generador*/

r1 = rand();
r2 = rand();
dado1 = (r1 % 6) + 1;
dado2 = (r1 % 6) + 1;
total = dado1 + dado2;
printf("primer dado = %d\n", dado1);
printf("segundo dado = %d\n", dado2);
printf("suma = %d\n", total);
return 0;
}
```

Para hacer este pequeño programa ha sido necesario considerar muchos detalles de bajo nivel (de abstracción). Entre estos detalles están:

- ¿Cómo se generan números aleatorios?. Consulta de manuales de referencia, etc.
- Funciones que lo hacen. Un sistema sencillo es mediante la función `rand()` que devuelve un `int`. Otro sistem es el uso de la función `random()`, que devuelve un `long` y permite una mayor calidad en la generación, aunque es más sofisticado y complejo, y tiene otras funciones asociadas.
- ¿Qué es una semilla?. Se establece con las funciones `srand()` o `srandom()`. Podemos usar el tiempo del sistema como semilla pero esto no es la mejor opción. Mejor leer un entero del fichero /dev/random (bloquea la lectura hasta

que haya suficiente entropía) o del fichero /dev/urandom (no bloquea aunque no haya suficiente entropía, pero es menos seguro). O usar hardware específico de generación de números aleatorios.

- El manual dice que la función *rand()* devuelve un valor entre 0 y RAND_MAX, entonces hay que normalizar el resultado devuelto para que esté entre 1 y 6. (El valor de RAND_MAX depende de la arquitectura, y es 2147483647 en i686 GNU/Linux kernel 2.6.27-7-generic)
- Y quizás otros detalles más. Qué estructuras de datos son adecuadas (aunque en este pequeño ejemplo este problema es muy sencillo), I/O específica de esta aplicación, formateo de la salida a pantalla del programa, etc.

Comentarios al ejemplo:

- Hacer hincapié en la cantidad de detalles de bajo nivel empleados.
- Utilización de manuales del lenguaje y del S.O.
- Todo este detalle de bajo nivel dificulta la labor del programador.
- Y eso que es solo un pequeño programa para el lanzamiento de dos dados. A mayor tamaño del proyecto software los detalles de bajo nivel se multiplican.
- Si hubiese que tener en cuenta todos los detalles de bajo nivel mientras se está desarrollando un gran proyecto software, la programación sería muy compleja. Y lo peor, sería también muy difícil que el software resultante fuese de calidad.

1.1.2. Un primer "vistazo" a la programación con TADs

- Los Ingenieros del Software ven como primer avance establecer una etapa inicial de **diseño** en el que se eviten los detalles de bajo nivel de la implementación.
- Esta etapa de diseño es una etapa preliminar de mayor nivel de abstracción.
- En esta etapa los protagonistas son los TADs, no las estructuras de datos ni el código de bajo nivel del programa.
- Esta etapa consiste en identificar y describir los TAD. Pero, ¿qué es un TAD?. Definiremos formalmente este concepto más adelante, ahora lo veremos directamente con un ejemplo.
- Como ejemplo veremos la resolución del problema anterior del lanzamiento de dos dados pero diseñando el TAD *Dados* para ello, del que escribiremos una breve descripción en lenguaje natural:

```
TAD Dados
    Representa el lanzamiento de dos dados.

OPERACIONES
    - lanzamiento: simula el lanzamiento y asigna nuevos valores
                a los dados.
    - getDado1: devuelve el valor del primer dado.
    - getDado2: devuelve el valor del segundo dado.
    - getSuma: devuelve la suma de los dos dados.
```

- Como se ve, hay una entidad de *alto nivel* que es el propio TAD Dados que representa el lanzamiento de los dados (desde luego que tendrá que almacenarlos de alguna forma, pero eso ahora no nos preocupa) y tiene una serie de operaciones.
- Entre las estructuras de datos que conocemos no hay ninguna que pueda mantener datos y operaciones. De ahí la invención de este nuevo concepto: el TAD.
- Incluso ya se puede obtener el programa final que resuelve el problema (aún sin tener terminado el propio TAD):

En C:

```
// juego.c
// segunda versión usando más abstracción pero sin clases

int main(void)
{
lanzamiento();
printf(‘‘dado1 = %d ’’, getDado1());
printf(‘‘dado2 = %d ’’, getDado2());
printf(‘‘suma  = %d ’’, getSuma());
}
```

Mejor en C++, ya que es más adecuado para la programación con TAD:

```
// dados.h
// La clase Dados representa el lanzamiento de 2 dados

class Dados{
private:
    int d1_, d2_;
public:
    void lanzamiento();
    int getDado1();
    int getDado2();
    int getSuma():
};

// juego.cc
// Usa la clase Dados y muestra resultados

int main(void)
{
Dados d;
d.lanzamiento();
cout << ‘‘dado1 = ’’ << d.getDado1();
cout << ‘‘dado2 = ’’ << d.getDado2();
cout << ‘‘suma  = ’’ << d.getSuma());
}
```

Comentarios sobre el ejemplo:

- Destaca la naturalidad y sencillez con la que se realiza el programa.
- Es importante describir bien su descripción y sus operaciones (más adelante

veremos que a esta descripción se le llama **especificación**).

- Es interesante entender el concepto de **usuario** o **cliente** de un TAD: el programador que posteriormente usará el TAD para hecer sus programas, y que no siempre es la misma persona que ha diseñado el TAD.
- Observar como ahora la función `main()` es entendible (y mantenible, etc.) fácilmente. Esto es porque el nivel de abstracción conseguido es alto, y los detalles de bajo nivel no intervienen en esta fase. Eso repercutirá en todos los factores de calidad del software desarrollado de esta forma (ver factores de calidad más adelante).

1.1.3. Encapsulado y ocultación de la información

- Un TAD tiene una **vista exterior**, en la que se ocultan detalles, estructuras de datos y los secretos de su comportamiento interno. Y también tiene una **vista interior**, en la que se puede ver cómo está hecha por dentro y como se comporta internamente cada operación, las estructuras de datos utilizadas, el código, etc.
- La separación clara de estas dos vistas es muy importante, ya que en el diseño de un TAD no debe intervenir la vista interna. Igual ocurre con el cliente del TAD, para el que no debe ser necesario conocer el interior (la vista interna) de un TAD para poder usarlo. Esto facilita el diseño y lo hace de más calidad, y también hace la labor del cliente más sencilla y potente.
- Esta separación vista interna/externa es como si la parte interna del TAD estuviera oculta o encapsulada con respecto a la vista externa. El encapsulado es un pilar de la programación con TADs que también recibe otros nombres: encapsulamiento, ocultación, etc.
- El TAD *Dados* oculta al diseñador y al cliente sus detalles. En sus respectivas tareas estos detalles no son relevantes, y de hecho, sería contraproducente dificultar sus respectivas labores con esos detalles irrelevantes. Estos detalles son irrelevantes a este nivel, cuando posteriormente haya que codificar el interior del TAD, estos detalles de la estructura de datos, el código, etc. pasarán a ser relevantes. Pero diferenciar ahora estos dos niveles es la base de la programación con TAD.
- No solo es cuestión de dificultad, muchas de las ventajas de la programación con TAD se pierden sin encapsulamiento. Por ejemplo, si el cliente conoce la vista interna y usa elementos de ella, cuando cambie esa vista interna, por ejemplo por cuestiones de eficiencia, el cliente deberá cambiar el código, con lo que el mantenimiento del sistema se vuelve costosísimo. Aparte de que si accede al interior del TAD puede provocar operaciones y usos no permitidos, errores, etc. Es mejor que el cliente se limite a conocer la vista externa y a usar **la interfaz del TAD**, es decir, las operaciones del TAD.

- Hacer hincapié en este aspecto con algún ejemplo más de la utilidad del encapsulamiento. El alumno debe buscar algún ejemplo más en la bibliografía o la literatura sobre este tema.

1.1.4. Lenguajes de Programación Orientados a Objetos y TADs

- Los LPOO son los que desarrollan mejor el paradigma orientado a objetos y la programación con TAD. Se puede programar con abstracción de datos en lenguages no orientados a objetos, pero es mucho más fácil y eficiente hacerlo en lenguajes orientados a objetos.
- Existen muchos y hay diferencias en el grado de desarrollo del paradigma orientado a objetos que realiza cada uno de ellos.
- Existen algunos lenguajes que suelen denominarse como LPOO "puros" o "genuinos": Eiffel, SmallTalk, Ruby, etc.
- Quizás el más utilizado sea C++, aunque algunos no lo consideran un buen LPOO puro como los anteriores.
- Hay muchos otros: Java, Pascal Orientado a Objetos, COBOL orientado a objetos, BASIC, VisualBASIC, Python, etc.

1.1.5. TADs en C++

Declaración del cuerpo de las funciones miembro de la clase *Dados*:

```
// dados.cc
// cuerpo de los métodos de la clase Dados
#include <cstdlib>
#include <ctime>

void Dados::lanzamiento()
{
int r1,r2;  // variables locales a la función
srand(time(NULL));
r1=rand();
r2=rand();
d1_=(r1%6)+1;  // d1_ y d2_ son var. privadas de la clase
d2_=(r2%6)+1;
}
int Dados::getDado1()
{
return d1_;
}
int Dados::getDado2()
{
return d2_;
}
int Dados::getSuma()
{
return d1_+d2_;
}
```

Conceptos iniciales de C++ necesarios para realizar la primera práctica de la asignatura:

- Notación C++ para inclusión de ficheros de cabecera.
- Guardas de inclusión (#define guards, headers guards).
- Espacio de nombres.
- Entrada y salida con cin y cout.
- El nuevo tipo "bool".
- El compilador gcc ahora como g++ (mismo gcc, GNU Compiler).
- Declaración de una clase.

- Qué ocurre al definir un objeto: reserva de memoria y el constructor.
- Las variables miembro son globales a todos los métodos de la clase.
- Acceso a miembros públicos (interfaz) y privados.
- Acceso prohibido a miembros privados para hacer efectivo el encapsulamiento.
- Introducción al *Unit Testing* y a Google Test (para la segunda sesión de prácticas).

1.2. El ciclo de vida del software

En este apartado se trata de situar el trabajo con TADs en el proceso global de desarrollo de un sistema software.

Es interesante el ejemplo de la denominada "crisis del software" a finales de los 60 y principios de los 70 (the difficulty of writing useful and efficient computer programs in the required time. The software crisis was due to the rapid increases in computer power and the complexity of the problems that could be tackled. With the increase in the complexity of the software, many software problems arose because existing methods were neither sufficient nor up to the mark. Fuente [http://en.wikipedia.org/wiki/Software_crisis]). Y de como esta crisis motiva la aparición del la Ingeniería del Software y de nuevos procesos y metodologías de desarrollo orientadas a una mayor calidad del software.

Uno de los modelos (o metodologías) clásicos de desarrollo de software dentro de la Ingeniería del Software es el modelo de desarrollo en cascada (*waterfall*) el cual, muy brevemente, comprende:

- Inicio: identificación de un problema. Decidir si es conveniente una solución software.
- Captura de requisitos: acopio de información del dominio del problema, expertos, clientes, usuarios, etc. Def. formal del problema.
- Análisis: analisis de los requisitos y toma de decisiones estratégicas, etc.
- Diseño: elaboración de un modelo que solucione el problema.
- Implementación (incluida la codificación).
- Prueba (incluida la corrección del código).
- Implantación (deployment).
- Documentación (durante todas las fases anteriores).
- Entrenamiento (formación del usuario) (no forma parte del desarrollo).

- Mantenimiento y soporte. Se estima que el 70 % del coste del software es de la fase de mantenimiento. En orden de frecuencia: cambios en los requisitos del usuario, cambio en el formato de datos, cambios de emergencia, fallos, cambio de hardware, cambio en la documentación, mejoras de eficiencia, otros.

Existen otras metodologías basadas por ejemplo en *Prototyping*, la creación rápida de prototipos iniciales y su evolución con la participación del usuario en el proceso de desarrollo, como son Rapid Application Development (RAD), Agile Software Development, Extreme Programming (XP), o métodos híbridos como Spiral, además de otros métodos (objetos de estudio en asignaturas como Ingeniería del Software).

No vamos a estudiar aquí cada una de estas metodologías. Nuestro interés en este punto es simplemente destacar que cuando trabajamos con TADs estamos realizando una tarea de diseño, al programar con TADs hacemos **diseño**, como se analiza en más detalle en el siguiente apartado.

1.3. El diseño del software. El diseño orientado a objetos

El diseño de software es un proceso y a la vez un modelo. Un proceso porque es una secuencia de pasos que permiten al diseñador describir todos los aspectos del software que se quiere construir. Un modelo ya que es el plan para construir el software, equivalente al plan de un arquitecto para construir un edificio. Comienza representando la totalidad de lo que se quiere hacer y poco a poco va definiendo como ir construyendo cada detalle. El modelo propuesto debe dar lugar a la consecución de un producto (el **software de calidad**) en el tiempo y con la satisfacción estimada.

Tras la captura de requisitos y el análisis comienzamos a diseñar un modelo que satisfaga las necesidades planteadas en las etapas previas.

El diseño orientado a objetos inicia entonces una serie de etapas centradas en identificar qué y quién dejando a un lado el cómo. Es ahí cuando empieza el diseño con TADs.

La programación con TADs es diseño puro y se deben tomar importantes decisiones de diseño que afectarán considerablemente a la calidad del producto, extensibilidad, etc. Entre otras:

- Qué actores, elementos, entidades y componentes compondrán el sistema, misión de cada uno. Los diferentes tipos/clases/categorías de dichos componentes.
- Arquitectura con la que se dotará el sistema y que relaciona a sus componentes.
- Patrones de diseño que se usarán (serán estudiados más adelante en la asignatura).
- Procedimientos y funciones importantes del sistema.
- Interfaz de las clases: nomenclatura, identificadores, parámetros, etc.

- Diseño de las pruebas del software.
- Código junto o en varias funciones, módulos, etc. (descomposición)(muy importante).
- Mantenimiento: refinando el diseño.

Otros elementos del diseño orientado a objetos:

- Estilo de la programación (convenciones internacionales y guías de estilo).
- El código fuente que creamos, su tipo, disposición, estructura, estética, orden, etc.
- Nombre de los identificadores: de clase, variables, de funciones, módulos, etc.
- Selección/diseño de los algoritmos y estructuras de datos críticas en el producto.
- Documentación y autodocumentación.
- Etc.

Como comentario sobre el que debatir, es interesante hacer ver al programador en formación que cada programa debe incorporar nuestro empeño, ilusión y sello personal de calidad. La elaboración de software es uno de los ejercicios más importantes de nuestra profesión.

Debemos poner mucho interés en hacer programas con un buen nivel de calidad. En este sentido, un gran obstáculo a la hora de elaborar programas de calidad, que no suele citarse en los libros de texto es la pereza del mal programador. En el momento de la programación el mal programador tiene pereza por: dedicar el tiempo previo necesario para hacer un buen diseño, autodocumentar bien los programas, tomarse un tiempo buscando buenos identificadores, separar y ordenar el código debidamente en distintos módulos con sus ficheros fuente, añadir comentarios adecuados, escribir el código con claridad, etc. Son tareas sencillas que aumentan enormemente la calidad de la programación y que su postergación (el "ya lo haré después...") solo tiene como resultado la pésima calidad de nuestros programas y desempeño de nuestra profesión.

Incluso se habla de los principios de todo programador (referencia al documento "Los Principios del Programador" de *Daniel Read*, que es una lectura recomendada (buscar en Google o en http://www.developerdotstar.com/mag/articles/read_princprog_espanol.html).

1.4. Software de calidad

La calidad de los programas, del software, es el objetivo fundamental de la abstracción de datos y del trabajo en esta asignatura. Pero qué es la calidad del software. En esta sección tratamos de sentar algunas bases acerca de este importante concepto.

En un primer enfoque podemos clasificar las cualidades del software en dos tipos: calidad funcional y calidad estructural.

La calidad funcional tiene que ver con el grado de acercamiento del funcionamiento del software a lo que se determinó en sus especificaciones y requisitos funcionales. Se trata de una característica que puede apreciarse durante la ejecución del software por parte del usuario, por ello se denominan también factor de calidad externo. Afecta de forma directa al usuario del software.

La calidad estructural tiene que ver con el cumplimiento de características internas no-funcionales del software que dan soporte a otras características internas (factor de calidad interno) del software también deseables o incluso más deseables como son la mantenibilidad, la facilidad de extensión, etc. Afecta de forma más directa al programador del software (y de forma indirecta usuario final).

Hay que resaltar que ambos tipos dependen del programador y que, al final, afectan directa o indirectamente al usuario final.

En cualquier caso el concepto de calidad del software no es fácil de definir. Existen incluso planteamientos que defienden que es imposible determinar la calidad de un software. A pesar de ello sí existen factores de calidad del software que se asumen en general como deseables y que son interesantes que, se use el modelo de desarrollo software que se use, estén presentes en el software resultante.

Los factores de calidad que nosotros estudiaremos están basados en el trabajo de McCall y colaboradores en 1977.

1.4.1. Operación del producto

Product operation. Características funcionales y operacionales.

Corrección

Correctness. Es un factor externo. Es la capacidad del producto para realizar de forma adecuada aquello para lo que fue creado, tal como se definió en los documentos de especificación y requerimientos.

Robustez

Reliability. Es un factor externo. Es la capacidad del producto de manejar correctamente situaciones imprevistas, de error, o fuera de lo normal. Por ejemplo cuando un programa puede hacer que todo un Sistema Operativo quede totalmente bloqueado, es porque ese Sistema Operativo no es muy robusto. Esta caraterística es imprescindible en sistemas como controladores de vuelo, pilotos automáticos, sistemas de seguridad, etc.

Eficiencia

Efficiency. Es un factor externo e interno. Se trata de conseguir que el programa, además de realizar correctamente aquello para lo que ha sido creado, lo realice

además de la mejor forma posible. Algunos factores que afectan a la eficiencia son: el espacio en memoria utilizado por el programa y el tiempo de ejecución (los dos más importantes en la mayoría de los casos), el espacio en disco, la memoria temporal, número de accesos a determinado hardware (de comunicaciones por ejemplo), etc.

Integridad

Integrity. Es un factor externo. El producto no debe corromperse por el simple hecho de su utilización masiva o por una gran acumulación de datos, o por operaciones complejas posibles, pero no previstas al cien por cien. Y protegido de accesos no autorizados. Y seguro.

Facilidad de Uso

Usability. Es un factor externo. Facilidad al introducir datos, interpretar datos, comprender errores, interpretar resultados,etc. Para usuarios con diferentes formaciones y aptitudes. Instalación, operación y supervisión. Capacidad multilingue.

1.4.2. Transición del producto

Product transition. Capacidad de adaptación a nuevos entornos.

Portabilidad

Portability. Es un factor externo e interno. Es la capacidad o facilidad del producto de ejecutarse en otro hardware diferente o en otro sistema operativo diferente. Aquí es muy importante que el programa no haga uso de características de bajo nivel del hardware o que aisle la parte dependiente del hardware para que al portarlo solo sea necesario modificar dicha parte.

Reutilidad/Reusabilidad

Reusability. Es un factor interno. Es la capacidad del producto de ser reutilizado en su totalidad o en parte por otros productos, con el objetivo de ahorrar tiempo en soluciones redundantes ya hechas con anterioridad. De esta forma, un programa debe agrupar en módulos aislados los aspectos dependientes de la aplicación particular, mientras que las utilidades que puedan generalizarse deben mantenerse preparadas para ser utilizadas por otros productos y así ahorrar costes de producción. Diseño por capas de diferente nivel de abstracción.

Compatibilidad

Interoperability. Es un factor externo. Es la facilidad que tienen los programas de combinarse entre sí. Es la posibilidad de utilizar los datos de salida de un programa como entrada de otro programa. La clave es la estandarización y el consenso entre

organismos, instituciones y empresas, en primera instancia, y entre programadores en última instancia.

1.4.3. Revisión del prodycto

Product revision. Aptitudes ante la modificación y el mantenimiento del producto.

Maintainability

Capacidad de encontrar y corregir un defecto en el software.

Extensibilidad

Flexibility. Es un factor interno. Es la facilidad de adaptar el producto a cambios en la especificación de requisitos. Cuanto mayor es el programa suele costar más esfuerzo, y si no se aplica un buen diseño, y sobre todo simple, este factor puede verse muy perjudicado.

Testability

Habilidad de poder validarse el software.

1.5. Otros factores importantes

Son factores que avanzan en importancia tanto como para ser considerados por separado.

1.5.1. Seguridad

Es un factor externo. Es la capacidad del producto de proteger sus componentes de usos no autorizados y de situaciones excepcionales de pérdida de información. Para ello debe prever mecanismos de control de acceso, encriptación, clave de acceso y también, desde el otro punto de vista copias de seguiridad, procesos rutinarios de comprobación, etc.

1.5.2. Accesibilidad

Es un factor externo.

Accesibilidad General: Tener acceso, paso o entrada a un lugar o actividad sin limitación alguna por razón de deficiencia, discapacidad, o minusvalía.

Accesibilidad Informática: consiste en el acceso a la información sin limitación alguna por razones de deficiencia, incapacidad o minusvalía. Por tanto, un programa accesible es aquél que permita el acceso a la información sin limitación alguna

por razón de deficiencia, incapacidad o minusvalía con el fin de poder elaborar, reproducir, manipular, etc. dicha información, así como al acceso sin ningún tipo de limitación por dichas causas a las herramientas y opciones de dicho programa.

1.5.3. Oportunidad

Capacidad de un sistema de ser lanzado cuando el usuario/mercado lo necesita, o antes. Ej: ideas "tontas"y "simples" han hecho milonario a muchos...

1.5.4. Economía

Los costes del producto. A veces, lo más importante de todo.

1.5.5. El compromiso entre factores de calidad

El compromiso entre ellos es necesario en muchas ocasiones y muchas veces no se puede cumplir alguno de los factores al cien por cien. Nuestra experiencia nos hace alcanzar ese compromiso para cumplir nuestras espectativas.

Nuestra tarea mientras nos formamos en programación es aprender métodos y herramientas que hagan que el software que desarrollamos tenga una alta calidad y por tanto debemos aprender como mejorar cada uno de los factores de calidad anteriores cuando programamos. Esa será una de las tareas fundamentales durante este curso.

1.6. Descomposición funcional frente a descomposición basada en objetos

En sus múltiples acepciones:

- Descomposición funcional, TDD (Top-Down Design), diseño descendente, arriba-abajo, descomposición descendente, *algorithmic decomposition, functional decomposition.*
- OOD (Object Oriented Design), diseño orientado a objetos, POO.

La descomosición funcional resulta poco adecuada para desarrollar sistemas de cierta envergadura (lo que sigue es del capítulo 5 de [Mey99]). Siendo un paradigma útil para pequeños programas y algoritmos individuales. Pero no se puede aplicar a sistemas software grandes y prácticos. El diseño descendente permuta comodidad a corto plazo por la inflexibilidad a largo plazo. Y se arriesga uno fácilmente a sacrificar la reutilización.

Las funciones son variables, cambian con el tiempo, se redefinen. Basar en ellas la descomposición es arriesgar mucho. En cambio los objetos, los datos son el candidato perfecto para ello. Aunque cambie el modo de procesar las nóminas de los trabajadores, seguirán usándose los datos que representan personas, horas trabajadas, escalas

salariales, etc., mejorándose la **extensibilidad**. Reutilizar una rutina es complicado porque se basan mucho en el acceso a los datos, son los propios datos y los módulos basados en ellos los que harán **más reutilizable** el software. Es mucho más difícil combinar funciones que objetos enteros. La compatibilidad se ve reforzada de forma natural.

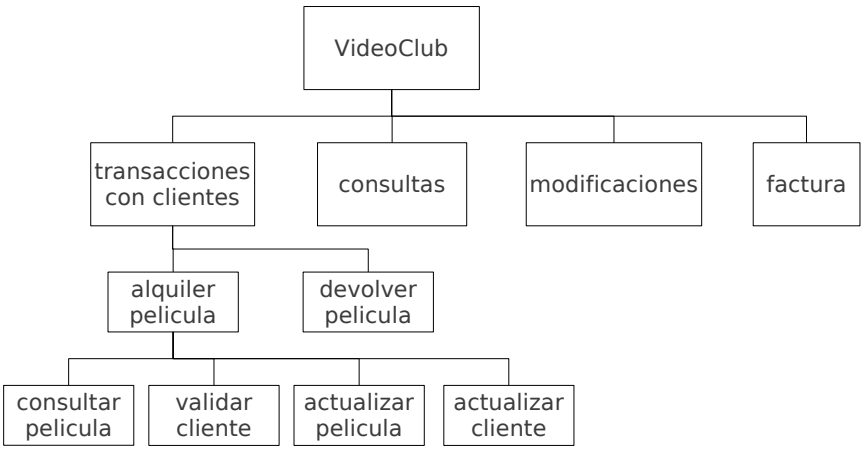
Extensibilidad, reutilización y compatibilidad son los tres pilares de la orientación a objeto.

Veamos un ejemplo con el que vamos a contrastar como se descompone un problema en TDD y en OOD. Es como si pensaramos en voz alta a la hora de hacerlo y van saliendo poco a poco los módulos.

1.6.1. Ejemplo Gestión de videoclub: Enunciado del problema

Construir un sistema software capaz de gestionar las operaciones básicas típicas de un videoclub. El sistema debe automatizar el proceso de alquiler de películas y el de su recepción una vez alquiladas, incluido el cálculo e impresión de la factura, que puede o no hacerse a la vez que la película es devuelta. El sistema debe permitir al operario el acceso y modificación de la información sobre películas y clientes.

1.6.2. Ejemplo Gestión de videoclub: Solución TDD



NIVEL 1: El stma. de alquiler se compone de los módulos: Transacciones con los clientes, Consultas, Modificaciones.

NIVEL 2: Transacciones con los clientes se compone de los módulos: Alquiler y Devolución.

NIVEL 3: Alquiler comprende las tareas: Consultar Película, Validar Cliente, Actualizar película, Actualizar Cliente.

NIVEL 3: Devolución comprende las tareas: Consultar Película, Validar Cliente, Acumular Factura, Actualizar Película, Actualizar Cliente.

(Ahora nos damos cuenta de que existe un nuevo módulo de NIVEL 2, ya que si acumulo la factura luego tendré que pagarla en algún momento.)

NIVEL 3: Pagar Factura comprende las tareas: Actualizar Cliente, Imprimir Factura.

NIVEL 2: Consultas se compone de los módulos: películas, Clientes.

NIVEL 2: Modificaciones se compone de los módulos: Añadir Cliente, Borrar Cliente, Añadir Película, Borrar Película, Modificar Cliente, Modificar Película.

(Se puede continuar, mediante refinamientos sucesivos, hasta concluir que los módulos son lo suficientemente simples para implementarlos directamente sin más análisis.)

Comentarios a esta forma de resolver el problema:

- ¿Qué ocurre cuando cambien los datos que maneja el sistema (películas, clientes, etc.)? ¿A cuántos módulos afectará?.
- ¿Cómo dividimos el trabajo entre varios programadores si todos tienen que saber de todo?
- Si nos fijamos todos los módulos son verbos, todo son funciones. ¿Es esta la forma más adecuada?. Ya lo iremos viendo...
- ¿Qué puedo reutilizar de lo que hay aquí en otra aplicación? (la verdad es que NADA o muy poco).
- Y ¿mantener este sistema?, ¿y modificarlo? ¿y ampliarlo, corregirlo, etc...?.
- La programación procedimental tiene como paradigma de programación: decida qué procedimientos necesita; utilice los mejores algoritmos que pueda encontrar. ([Sch04] pág. 23).
- Si una vez hecho el programa, quisiéramos que el establecimiento también vendiera, por ejemplo, libros o cualquier otro producto, ¿en qué afectaría este cambio al código ya desarrollado?. Lo cierto es que se podría reutilizar poco. Habría que rediseñar desde el principio el sistema, habría que añadir una nueva raíz al árbol para los productos nuevos. El nuevo NIVEL 0 quedaría como sigue. NIVEL 0: módulo para gestión de vídeos y módulo para gestión de libros. Sería un cambio muy dramático y costoso.

1.6.3. Ejemplo Gestión de videoclub: Solución OOD

Ahora veamos como se hace un diseño orientado a objetos. Pueden utilizarse las siguientes estrategias:

- Lo primero es identificar los actores del problema, los objetos, o mejor, las clases de objetos, o por simplificar, las clases.
- Correspondencia directa: clases **cliente** y **película**.
- Simulación de escenarios: el cliente devuelve la película, se escanea el código de barras.... Aparece la clase para controlar el *scanner*.
- Similitud hardware: clases **scanner** e **impresora**.
- Uso de las clases en pequeños programas que solucionen operaciones concretas.

Las principales fuentes de objetos son:

1. Correspondencia directa.
2. La reutilización de objetos ya creados.
3. La experiencia y la imitación como método para que el desarrollador cree nuevos objetos.

```
Algoritmo Alquiler:
===================
Pelicula pelicula;
Cliente cliente;
FicheroClientes fichClientes;
FicheroPeliculas fichPeliculas;
ScannerMano scanner;

int id;
id = scanner.leerCodigoPelicula(); //El cliente lleva la película
                                   //de la estanteria al mostrador

pelicula = fichPeliculas.recuperaPelicula(id);

id = scanner.leerCodigoTarjeta();

cliente = fichClientes.recuperaCliente(id);

if (cliente.comprobarOK())
   {
```

```
   cliente.reservaPelicula(pelicula);
   pelicula.estaOcupada();
   }

fichPeliculas.actualizaPeliculas(pelicula);
fichClientes.actualizaCliente(cliente);


Algoritmo Devolución:
=====================
pelicula = fichPeliculas.recuperarpelicula(scanner.leerCodigoCinta());
cliente = fichClientes.recuperaCliente(scanner.leerCodigoTarjeta());

cliente.devuelve(pelicula);
cliente.actualizarFactura();
pelicula.estaDisponible();

fichPeliculas.actualiza(pelicula);
fichClientes.actualiza(cliente);
```

Comentarios a esta forma de resolver el problema:

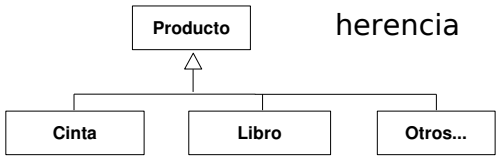
- Con este ejemplo vamos a contrastar como se descompone un problema en TDD y en OOD. Es como si pensaramos en voz alta a la hora de hacerlo y van saliendo poco a poco los módulos. Pelicula, Cliente, FicheroClientes, FicheroPeliculas, ScannerMano, Impresora.
- Las clases son independientes en cuanto a su comprensión, desarrollo, mantenimiento, etc.
- Las clases Scanner e Impresora pueden servir directamente para otras aplicaciones.
- Los programas son más naturales, más fáciles de entender y, por tanto, de hacer y de mantener.
- La POO tiene como paradigma de programación: decida qué clases necesita; ofrezca un conjunto completo de operaciones para cada clase; haga explícito lo que hay en común mediante el uso de la herencia (ya lo veremos). ([Sch04] pág. 40)

A pesar de que el concepto de herencia lo vamos a definir y desarrollar en más profundidad más adelante en el curso, haremos aquí algunos comentarios sobre él ya que es importante para ver que la POO permite una gran reusabilidad del código, precisamente gracias al concepto de herencia. Los siguientes puntos se refieren a

la herencia y deben releerse (y se entenderán mejor) cuando este concepto se haya trabajado en profundidad.

- Otro comentario interesante aquí, relacionado con la reusabilidad comparando la POO con otras metodologías, es qué ocurriría si este videoclub ampliara la oferta de productos y pasara a ofrecer además libros, por ejemplo. Este nuevo requisito sería acogido mucho menos dramáticamente en la versión orientada a objetos. Sería mucho más fácil modificar y ampliar el programa orientado a objetos. Piénsese como podría plantearse este nuevo requisito. Solo habría que crear la clase Producto y que de ella derivaran todas las clases de productos que se ofertan en la tienda. Los procedimientos sobre objetos de la clase *Producto* serían exactamente los mismos para todos ellos.

- El caso del punto anterior viene a ilustrar el concepto de **herencia** de la POO. El mecanismo de la herencia es uno de los pilares de la reutilización en POO, y permite, a partir de una clase, derivar otra nueva clase parecida aportando únicamente el código que las diferencia, reutilizándo así todo el código de la clase base.



- La necesidad de que los diseños sean abiertos (a nuevas modificaciones, ampliaciones, etc.) es un concepto natural para los desarrolladores de software, ya que éstos saben que es casi imposible prever todos los elementos, datos, operaciones, etc., que un módulo necesitará durante su tiempo de vida. Por tanto, los programadores desearán retener tanta flexibilidad como sea posible en los módulos para futuros cambios y extensiones (ver algo más sobre esto en los *Cinco principios en la construcción de Software* [Mey99] pág. 55).

  A la vez que hacer un diseño abierto, los desarrolladores desearán que sea cerrado en el sentido de que tienen que terminar el proyecto a tiempo. Ambos requisitos son difíciles de conseguir con técnicas tradicionales. Supongamos que el módulo A debe considerar ahora nuevas extensiones. Mediante las técnicas tradicionales tenemos las opciones de modificar dicho módulo (con lo que los antiguos programas que dependían de él pueden dejar de funcionar), o bien hacer una copia del módulo A en un módulo A' modificado. Con lo cual tendríamos el inconveniente de que ahora habría dos módulos iguales pero diferentes que mantener, es decir, el doble de esfuerzo.

La POO nos permite, mediante el mecanismo de la herencia, derivar un nuevo módulo del módulo A solo con las aportaciones o extensiones necesarias. Esto es más eficiente y menos costoso de mantener (hay una discusión sobre este tema en la sección *Cinco principios en la construcción de Software* de [Mey99] pág. 55).

1.6.4. Ejemplo Gestión de videoclub: Conclusiones

- La POO parece ser una forma más natural de diseñar programas. El proceso es más sencillo.
- La POO posee mayor nivel de abstracción que permite analizar el problema de forma más cómoda y accesible a cualquiera.
- La POO demuestra mayor facilidad de impregnar de calidad los desarrollos software.
- La TDD es utilizada aun mucho y en muchos casos con éxito, pero requiere un mayor esfuerzo en todas sus etapas y no aporta tantos beneficios.
- El simple hecho de usar TADs, ya hace que aumente la calidad de nuestros programas. Si, además, se hace un buen uso de ella, eso ayuda bastante más. Aprendamos pués a usarla.
- Si la tienda decidiese ampliar los productos a disposición del cliente, sería simple ampliar el programa con OOD ya que solo implicaría añadir las clases nuevas derivadas que se correspondan con los nuevos productos, sin embargo si se usa un programa usando TDD la ampliación sería más compleja, ya que los nuevos productos tendrían un nivel jeráquico similar al de las cintas, y aparecería un nuevo módulo al mismo nivel que el del videoclub, sin poder reusar casi nada.

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.