75.41 Algoritmos y Programación II
Tipo de Dato Abstracto

Dr. Mariano Méndez[1]

[1]Facultad De Ingeniería. Universidad de Buenos Aires

5 de mayo de 2020

1. Introducción

Esta sección está dedicada a unos de los conceptos más importantes en lo que respecta al ámbito de la informática y en este caso más precisamente a de la programación, ese concepto es la *Abstracción*. La Abstracción es el proceso por el cual se despoja a un problema o cosa de la complejidad que no es relevante para la solución de un problema determinado. Los lenguajes de programación de alto nivel como por ejemplo C, C++, Python, Java por nombrar algunos poseen un conjunto de herramientas que hacen extensible al lenguaje para construir nuevas abstracciones según los programadores lo requieran [**?**].

Cuando se habla de abstracción lo que se espera "es un mecanismo que permita la expresión los detalles relevantes y la supresión de los detalles irrelevantes" [**?**]. Siguiendo esta afirmación, en lo que respecta a la programación lo que respecta al uso del qué va a ser realizado por la abstracción es relevante, y por otro lado el cómo va a ser realizado es irrelevante.

Específicamente para el lenguaje de programación C las herramientas que hacen extensible al lenguaje con nuevas abstracciones son:

- funciones y procedimientos
- archivos de encabezado
- tipos de datos primitivos del lenguaje

1.1. La Abstracción

Abstracción proviene del latín abstracto y está vinculado al verbo abstraer que significa: **separar aisladamente en la mente las características de un objeto o un hecho, dejando de prestar atención al mundo sensible para enfocarse solo en el pensamiento**. Una de las mejores imágenes que describe el concepto de abstracción es la pintura de Picasso llamada el toro.

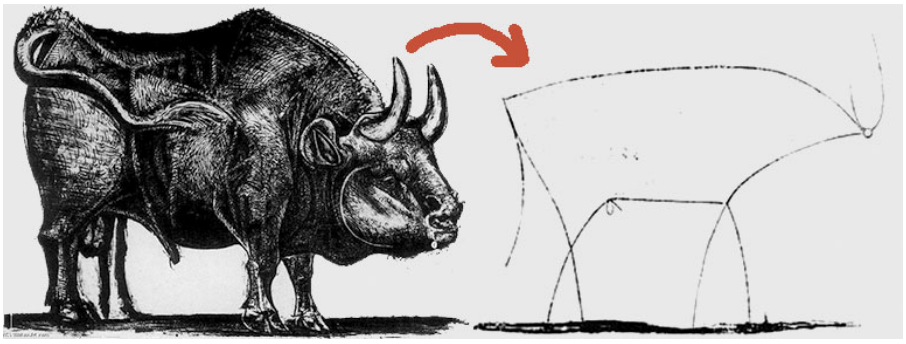

Figura 1

De acuerdo con la matemática, la abstracción es el proceso intelectual a través del cual separamos mentalmente las cualidades particulares de varios objetos para fijarnos únicamente en uno o diversas características comunes. Es a través del rigor, que se logra esta operación mental denominada generalización simple.

Una de las definiciones y usos de abstracción se puede ver en el siguiente chiste: Un biólogo, un matemático y un físico discuten del motivo por el cual el guepardo es tan veloz. El Matemático dice: -Pff es trivial la explicación,

si tomamos la función guepardo(x) y la integramos entre el hocico y la cola, se explica perfectamento por qué el guepardo se mueve con esa velocidad... -Que disparate. Dice el Biólogo: - Esto se explica por los miles de años de evolución, por la teoría darwinista de la especialización de las especies, la adaptación del aparato muscular, bla bla bla.. El físico que observaba callado, dice: -Si tomamos al guepardo puntual ....

1.2. Los Tipos de Datos

Un tipo de dato define dos cosas importantes:

1. Al conjunto de todos los valores posibles que una variable de ese tipo puede tomar.
2. Las operaciones que las variables de ese tipo de dato pueden utilizar.

Es normal que los lenguajes de programación tengan *tipos primitivos (predefinidos) de datos*. Un tipo de dato es primitivo cuando no debe especificarse información extra para definir una variable de ese tipo o sobre las operaciones que pueden hacerse sobre las variables. Por ejemplo, en el lenguaje C el tipo de dato *int* o entero:

Tipo	Conjunto de Valores	Operador	Operación	Resultado
int	-2^{31} y $(2^{31}-1)$ implementado como complemento a 2	+ (suma)	$5+3$	8
		- (resta)	$5-3$	2
		* (multiplicación)	$5*3$	15
		/ (división entera)	$5/3$	1
		% (resto)	$5\,\%3$	2

1.3. ¿Qué es un Tipo de Dato Abstracto?

Por un lado, cuando un programador utiliza una función, a este debería importarle únicamente en qué es lo que hace la función – por ejemplo, cuando se utiliza printf nadie se pregunta cómo está implementada, sino uno se centra en que esta imprime un valor determinado por pantalla– es decir en lo que esta le provee, es decir, lo importante de la abstracción es el uso que se hace de ella y no como la abstracción se implementa.

Por otro lado las funciones y procedimiento permiten descomponer los problemas en sub-problemas, haciendo que partes de las áreas se realicen en las distintas funciones que se llaman unas a otras y desde el programa principal. Esta herramienta apunta a capturar el concepto de abstracción pero no es suficiente por sí sola.

Un **Tipo de Dato Abstracto (TDA)**, según Liskov y Zilles, define una clase de objetos abstractos los cuales están completamente caracterizados por las operaciones que pueden realizarse sobre esos objetos. Esto quiere decir que un tipo de dato abstracto puede ser definido describiendo las operaciones características para ese tipo. En otras palabras, un tipo de dato abstracto está definido no sólo por una**estructura de datos** sino también por las **operaciones** que se define sobre esa estructura, es decir que son ambas cosas juntas.

Cuando un programador hace uso de un Tipo de Dato Abstracto (TDA), este se interesa únicamente en el comportamiento que tal objeto exhibe y no en los infinitos detalles de cómo se llegó a su Implementación. Más precisamente la visión de aquel que utiliza un TDA es exclusivamente de **Caja Negra**. El comportamiento de un TDA es capturado por el conjunto operaciones características o primitivas. Cualquier detalle de Implementación sobre el TDA es innecesario para aquel que hace uso del TDA.

Los TDA deben tender a parecerse a un tipo de dato primitivo provisto por el lenguaje de programación. Uno usa variables de tipo entero y no se pregunta como la computadora implementa internamente ese tipo, utiliza sus operaciones características o primitivas como la suma, división entera, resto, multiplicación, resta y resto de la división entera. En el caso de los TDAs el programador (usuario) está haciendo uso del concepto o abstracción que logra el TDA en un determinado nivel, pero no de los detalles de bajo nivel de cómo se realizan (implementación del complemento a 2).

Una importante implicación del uso de un tipo de datos abstracto, es que la mayoría (por no decir todas) las operaciones que se utilizan en un programa de un determinado tda pertenecen al set de operaciones características o primitivas del mismo.

2. El Qué y el Cómo

Las visiones de caja negra y caja blanca, hacen una importante diferencia:

- El **Qué**: cuando se habla del qué se está haciendo referencia al concepto de funcionalidad. La funcionalidad está estrechamente relacionado a la pregunta ¿Qué hace esto? Es el concepto de **caja negra**.

- El **Cómo**: cuando se hace referencia al cómo se está hablando de la forma en que algo está diseñado o implementado. Cuando se habla de diseño o implementación se está haciendo referencia a la estructura interna o a la forma en la cual la funcionalidad de algo es llevada a cabo. Por ello está estrechamente ligado a la pregunta de ¿como lo hago? Concepto de **caja blanca**.

En el estudio e implementación de los tda poder separar correctamente el **qué** del **cómo** es fundamental. Por ejemplo, un string es un tipo de dato muy común en los lenguajes de programación de alto nivel. Cuando se pregunta **qué** es un string, una respuesta acertada seria: es un tipo de dato en el cual se pueden almacenar palabras, frases o textos.

Cuando se pregunta de **cómo** se implementa el tipo de dato string, las respuesta pueden ser muy variadas. Existen diversas implementaciones. Por ejemplo en el lenguaje de programación C un string es una arreglo de caracteres cuyo contenido válido está delimitado por un \0 , ver figura 2. En Pascal un string es un arreglo de caracteres cuya contenido válido está informado en la posición 0, ver figura 3. Otra forma posible de implementar un string seria con un registro de dos campos uno que posea una cadena de caracteres y el otro la longitud, ver figura 2:

H	O	L	A	∅	M	U	N	D	O	\0

Figura 2: Implementación de un String en C

10	H	O	L	A	∅	M	U	N	D	O

Figura 3: Implementación de un String en Pascal

H	O	L	A	∅	M	U	N	D	O
10									

Figura 4: Pila de expedientes

3. Un Qué y Mil Cómos

Si bien el título de esta sección puede parecerse a una canción de Sabina, centraliza bien la idea de como debe pensarse un tda. A partir del momento que se comienzan a manejar estructuras de datos complejas, es importante dividir dos conceptos básicos:

- Funcionalidad (de parte del qué)
- Diseño e Implementación (de parte del cómo)

Un tda debe pensarse siempre desde dos aspectos. El primero, desde la visión del implementador del tda este deberá basarse en la forma en que se diseña e implementa el tda, en base a la funcionalidad que el mismo debe cumplir. La segunda, desde la visión del usuario del tda, al usuario del tda únicamente le importa la funcionalidad del mismo, es decir *que haga lo que dice que hace*. Estas dos visiones son como dos conjuntos disjuntos, no tienen nada en común.

Ambas partes están obligados **a cumplir un contrato**, el mismo está implícitamente aceptado en el archivo cabecera (tda.h) del tda. Para ello es de vital importancia entender el proceso general cuando se trabaja con tdas.

En una primera etapa E. Dijkstra implementa un tda que tiene una determina funcionalidad (un determinado qué) que viene dado por la especificación de los datos que componen el tda y las operaciones sobre los mismos. Además Dijkstra diseña su propia implementación de las miles de formas de hacerlo (mil cómos) este elije el diseño que le parece mejor:

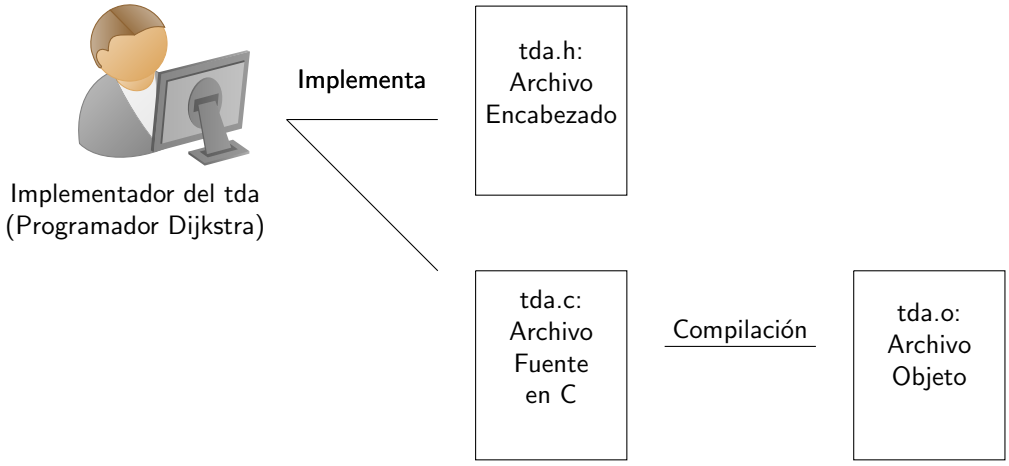

Figura 5: Implementación de un tda

Una vez que el el tipo de dato abstracto está implementado, el mismo puede ser utilizado por terceras partes, es decir por otras personas. En este caso por D. Knuth un programador que necesita utilizar el tda implementado por E. Dijkstra. Lo único que el implementador debe proporcionarle al usuario del tda es el archivo cabecera (tda.h) y el archivo objeto (tda.o).

Es importante familiarizarse con ciertos términos que se utilizan cuando se hace referencia a los tda:

1. **Implementar**: Poner en funcionamiento o llevar a cabo una cosa determinada. Está íntimamente ligado con el **Cómo**.
2. **Implementación**: Una forma de hacer algo.
3. **Implementador**: Aquel que implementa o construye algo.
4. **Funcionalidad**: Conjunto de características que hacen que algo sea práctico y útil. Íntimamente relacionado con el **Qué**.
5. Diseño: Actividad creativa y técnica encaminada a idear objetos útiles y estéticos que puedan llegar a producirse. También aplica a la informática referiendose a la forma en que fueron pensadas las implementaciones.

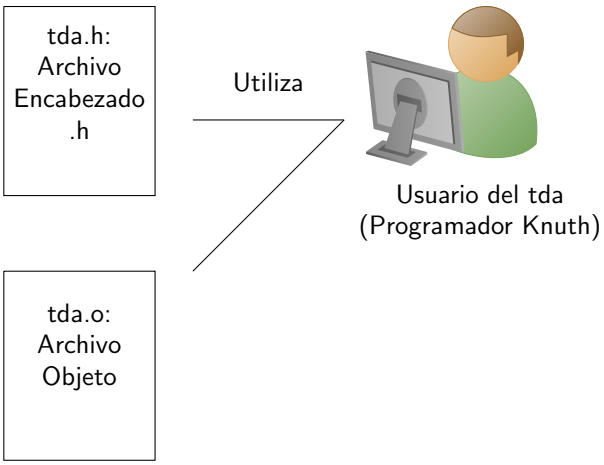

Figura 6: Utilización del tda

Básicamente existe un contrato tácito entre el implementador del tda y sus usuarios. El implementador debe garantiza que la funcionalidad expuesta en el archivo cabecera (.h) debe garantizarse. Es mas la utilización de **Pre-Post** condiciones es **fundamental** para este fin.

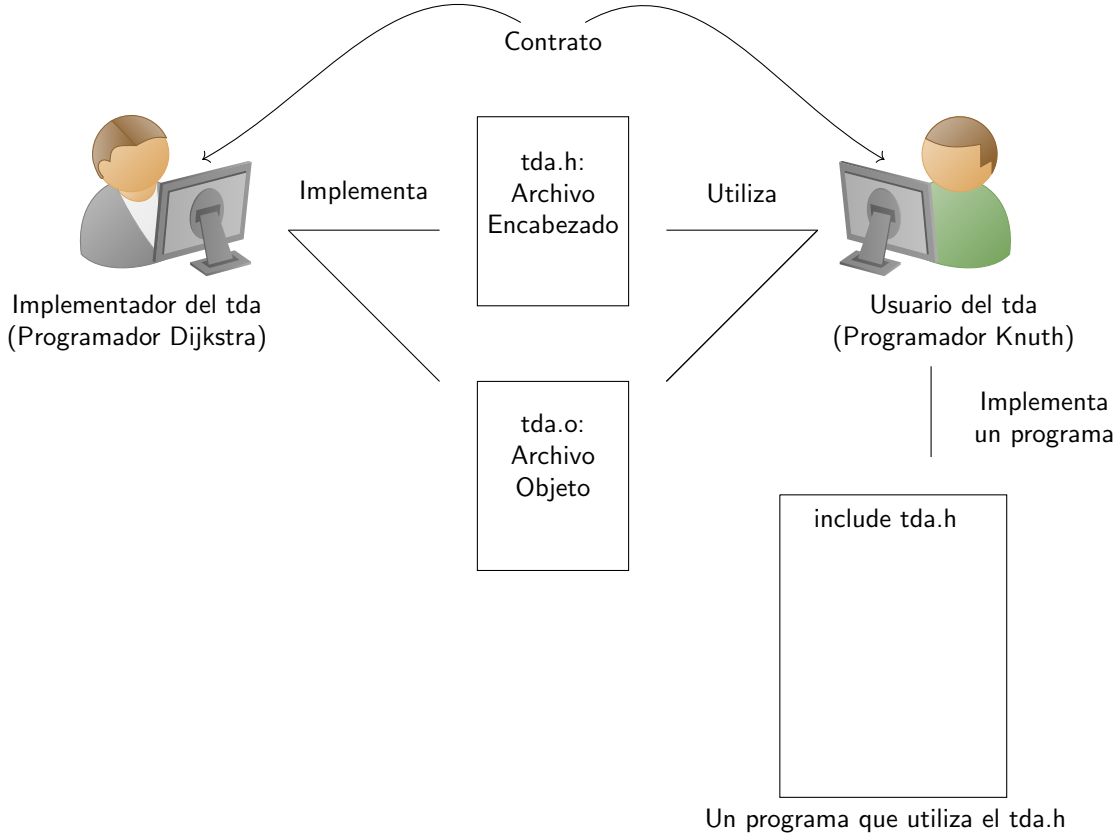

Figura 7: Contrato entre Implementador y Usuario de un tda

4. Un Clásico

El tda Número Complejo El el típico ejemplo basado en la descripción y utilización de un tipo de dato simple, que la mayoría de los lenguajes no posee como tipo primitivo (Excepto Fortran por ejemplo). Para ello se utiliza el tipo de dato primitivo double, un struct y las operaciones definidas para los números complejos

4.1. El Header de Dijkstra:

```c
#ifndef COMPLEJO_H
#define COMPLEJO_H

#include <stdio.h>

typedef struct complejo {
    double real;
    double imag;
} complejo;

void crear_c(complejo * c, double real, double imag);

/* c = a + b */
void  add_c( complejo * c, complejo  a, complejo  b );

/* c = a - b */
void  sub_c( complejo * c, complejo a, complejo b ) ;

/* c = a * b */
void mul_c( complejo *c, complejo a, complejo b ) ;
/* c = a / b */

void div_c( complejo *c,complejo a,complejo b ) ;

/* read a complex number from stdin */
void read_c( complejo * c );

```

```
/* write a complex number to stdout */
void print_c( complejo c ) ;

#endif
```

4.2. La Implementación de Dijkstra:

```
#include <stdio.h>
#include "complejo.h"

void crear_c(complejo * c, double real, double imag){
        c->imag=imag;
        c->real=real;

}

/* c = a + b */
void  add_c( complejo * c, complejo  a, complejo  b ) {
   c->real = a.real + b.real;
   c->imag = a.imag + b.imag;
}

/* c = a - b */
void  sub_c( complejo * c, complejo a, complejo b ) {
   c->real = a.real - b.real;
   c->imag = a.imag - b.imag;
}

/* c = a * b */
void mul_c( complejo *c, complejo a, complejo b ) {
   c->real = a.real * b.real - a.imag * b.imag;
   c->imag = a.imag * b.real + a.real * b.imag;
 }

/* c = a / b */
void div_c( complejo *c,complejo a,complejo b ) {
   double  _abs_sq = b.real * b.real + b.imag * b.imag;
   c->real = ( a.real * b.real + a.imag * b.imag ) / _abs_sq;
   c->imag = ( a.imag * b.real - a.real * b.imag ) / _abs_sq;
}

/* read a complex number from stdin */
void read_c( complejo * c ){
        // scanf( "%lf%lf", &c.real, &c.imag );
}

/* write a complex number to stdout */
void print_c( complejo c ) {
   printf( "%f + %fI", c.real, c.imag );
}
```

4.3. El Programa de Knuth:

```
#include <stdio.h>
#include "complejo.h"



int main(){
        complejo c1,c2,c3;

        crear_c(&c1,2,1);
        crear_c(&c2,0,1);
        add_c(&c3,c1,c2);
        print_c(c3);
        return 0;
}

```

5. Un Ejemplo Práctico

la Secretaria de Transporte planea realizar un sistema de boleto electrónico, todos los transportes tendrán una maquina que descontará de una tarjeta precargada el monto del viaje. Se podrá recargar la tarjeta y consultar su saldo.

Cómo siempre se debe realizar un análisis del problema que quiere resolverse. En este caso se quiere que una determinada entidad llamada sube (una tarjeta) pueda reemplazar al boleto, ser utilizada en cualquier transporte, la misma llevara un saldo de dinero cargado, y cada viaje descontaré el valor que corresponda a dicho saldo.

Como implementador del tda se debe determinar cuales serán las operacioines y datos que tendrá el mismo. Haciendo un análisis rápido el tda SUBE podría almacenar el **documento** del usuario y el **saldo** de la tarjeta. Por otro lado las operaciones que se deberían poder realizar sobre una tarjeta SUBE serían:

- crear: esta operación deberia generar una tarjeta SUBE válida.
- destruir: esta operación debería eliminar una tarjeta SUBE.
- cargar: esta operación permite cargar saldo en pesos a la tarjetaSUBE.
- realizar-viaje: esta operación permite descontar el importe de un viaje a la tarjeta SUBE
- saldo: esta operación devuelve el importe del saldo de la tarjeta SUBE.
- propietario: devuelve el dni del propietario dela tarjeta SUBE

5.1. El Contrato

Una vez definidos los datos y las operaciones se deben establecer el contrato, para ello se crea el archivo cabecera, en este caso sube.h:

```
#ifndef SUBE_H
#define SUBE_H

#define MAX_VIAJES 3

typedef struct Sube Sube;

Sube * sube_crear(const char dni[8],  double saldo_inicial );

void sube_cargar(Sube* tarjeta, double monto);

double sube_saldo(Sube* tarjeta);

const char *  sube_documento( Sube * tarjeta);

void sube_realizar_viaje(Sube* tarjeta, double costo);

 void sube_destruir( Sube * tarjeta);

#endif
```

Una característica a destacar es que no se ha definido que es el struct Sube, y sin embargo es posible utilizarla sin haberla definido. Esta es una estrategia para que la implementación del struct sube no sea necesaria conocerla para poder utilizarla.

Ya definido el tda con sus operaciones y el contrato de uso del mismo, es decir, su **funcionalidad** ahora hay que implementar de alguna forma la misma en el archivo que lleva su mismo nombre pero .c, en este caso sube.c:

```

#include<time.h>
#include<stdlib.h>
#include<string.h>
#include "sube.h"

struct Sube {
    char    dni[8];
    double  saldo;
};


Sube * sube_crear( char dni[8], double saldo_inicial){
    Sube* tarjeta;

        tarjeta= (Sube *) malloc(sizeof(Sube));

    tarjeta->saldo = saldo_inicial;
    strcpy(tarjeta->dni,dni);

    return tarjeta;
}
```

```
23
24 void sube_cargar(Sube* tarjeta, double monto){
25     tarjeta->saldo += monto;
26 }
27
28 double sube_saldo(Sube* tarjeta){
29     return tarjeta->saldo;
30 }
31
32 const char * sube_documento( Sube * tarjeta){
33         return tarjeta->dni;
34 }
35
36
37 void sube_realizar_viaje(Sube* tarjeta, double costo){
38
39     tarjeta->saldo -= costo;
40 }
41
42 void sube_destruir( Sube * tarjeta){
43         free(tarjeta);
44 }
```

Una vez terminada la implementación, solo falta compilar el .h para generar el .o y así poder distribuirlo.

6. El Proceso de Desarrollo de Software

Los programas y las bibliotecas que se construyen en lenguaje C forman parte del concepto de **Software**. La definición de este concepto es muy variada:

- Es el conjunto de los programas de cómputo, procedimientos, reglas, documentación y datos asociados, que forman parte de las operaciones de un sistema de computación. (IEEE estandar 729).
- Es todo aquello que no es hardware.
- el software es toda la información procesada por los sistemas informáticos: programas y datos.

El software no se fabrica, el software se desarrolla. El desarrollo de software no es una única tarea sino que incluye a un conjunto de etapas, muy parecidas a las utilizadas para la resolución de problemas computacionales. Este conjunto como mínimo debe comprender las siguientes tareas:

- **Análisis**: en esta etapa el foco se pone en la comprensión del problema, determinar cuales son aquellas cosas que se requieren para la resolución del mismo, etc. El análisis está muy ligado a ¿**Qué** es lo que hay que hacer?. poder responder a esa pregunta con claridad es un avance importante dentro del proceso de desarrollo.
- **Diseño**: La etapa del diseño se basa en el el proceso de crear una solución teniendo en cuenta múltiples aspectos del software y también del hardware sobre el cual se va a ejecutar el software que está siendo diseñado. El diseño pone su interés en el **Cómo**. Está muy relacionado con el problema, el usuario y la solución.
- **Implementación**: Esta etapa es la más conocida por los programadores pues es la etapa en la que a partir de un análisis acabado del problema, de los algoritmos a ser utilizados, las estructuras de datos, etc. se desarrolla el software.
- **Pruebas**: No puede haber software andando si previamente a su distribución este no ha sufrido una serie de baterías de pruebas para asegurarse, que por lo menos, lo que ha sido probado funcione.
- **Instalación**: Esta etapa se basa en hacer que el software esté andando.
- **Mantenimiento**: Como el software no es un producto que una vez acabado no sea susceptible al cambio, a arreglos, a mejoras o incluso a reparación. La etapa de mantenimiento justamente se ocupa de esto.

Es decir que para desarrollar software hacen falta realizar ese conjunto de tareas y en ese orden.

7. Ventajas de la Utilización de Tdas

La utilización de tda tiene muchas ventajas a la hora de realizar programas de gran tamaño.

7.1. Manejan la Abstracción

Los tdasson la herramienta perfecta para manejar la abstracción. La abstracción permite simplificar la realidad mediante el despojo de complejidad que no es inherente al problema en estudio. Es más los tdas permiten crear niveles de abstracción basándose en construir tdas nuevos usando tipos primitivos y otros tdas.

7.2. Encapsulamiento

Es la propiedad por el cual un tda debe exponer la menor cantidad posible de información del *cómo* está implementado, y debe por ende hacer que el usuario del tda se base en las funciones que el mismo entiende. El implementador de un tda debe poder ocultar la mayor cantidad de detalles sobre la **implemntacion** y el diseño del mismo.

7.3. Localización del Cambio

Cuando existe un error dentro de un programa es mucho más fácil detectarlo pues la utilización de los tdas fuerza la *modularización*

Referencias