

PROGRAMACIÓN ORIENTADA A OBJETOS (POO)

OBJECT ORIENTED PROGRAMMING (OOP)

curso 24/25

POO: CURSO DE C++

- 1 Introducción a la POO y a C++. Compilación con gcc/g++
- 2 Clases y Objetos.
- 3 Referencias y usos de 'const'. Funciones inline
- 4 Herencia y herencia múltiple.
- 5 Sobrecarga de funciones y sobrecarga de operadores. Funciones friend.
- 6 STL
- 7 Range for y deducción automática de tipo con auto.
- 8 Enumeraciones con enum
- 9 Herencia public, private y protected
- 10 Inicializadores y constructores de copia
- 11 Punteros a objetos. Funciones virtuales. Polimorfismo estático (en tiempo de compilación). Polimorfismo dinámico (en tiempo de ejecución) o vinculación dinámica.
- 12 Plantillas de función y plantillas de clase.
- 13 Manejo de ficheros en C++. Reserva de memoria con new y delete
- 14 Insertadores y extractores propios

POO: TEMAS DE TEORÍA

TEMA 1: ABSTRACTION & SOFTWARE DESIGN

TEMA 2: SOFTWARE DE CALIDAD

TEMA 3: DESCOMPOSICIÓN MODULAR

TEMA 4: TDD vs OOD. ESPECIFICACIÓN E IMPLEMENTACIÓN

TEMA 5: REUTILIZACIÓN

TEMA 6: 4 PILARES DE LA TECNOLOGÍA OO

TEMA 7: PATRONES DE DISEÑO

TEMA 1

ABSTRACTION & SOFTWARE DESIGN

ABSTRACTION

¿QUÉ ES?

... simplificar (hacer fácil de entender) un sistema complejo...

ABSTRACTION

¿POR QUÉ? ¿PARA QUÉ HACER ABSTRACCIÓN?

- Porque un sistema software es un sistema complejo y necesitamos hacerlo fácil.
- También facilita la calidad.

Quizá el concepto más importante de la POO

ABSTRACTION

¿CÓMO SE HACE?

Añadiendo una capa de abstracción sobre la complejidad interna de cada módulo software

ABSTRACTION

- Definiciones:
 - **DRAE**: *Abstraer es 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.*
 - **Computer Science**: *reducir la complejidad de un módulo:*
 - 1.- **Ocultando** *detalles innecesarios/irrelevantes, y*
 - 2.- **Destacando** *los detalles esenciales/relevantes*

ABSTRACTION

- La H^a de la informática es la sucesiva adición de capas de abstracción:
 - *More abstraction on-top of complex systems*

ABSTRACTION. EXAMPLE A

Lanzar unos dados

```
#include <time.h>
#include <stdlib.h>
int d1, d2, sum, r1, r2;

srand(seed); // pseudo-random integer seed
r1 = rand();  // in the range 0-RAND_MAX
r2 = rand();
d1 = (r1 % 6) + 1;
d2 = (r2 % 6) + 1;
sum = d1 + d2;
```

¿Es este código entendible
fácilmente, modificable, ...
mantenible?

Necesitamos mayor nivel de abstracción para
enfrentarnos a problemas grandes

Para ello:

- Debemos hacer un buen diseño de nuestras abtracciones.

DE LA ABSTRACCIÓN AL... DISEÑO

1. Analizar un problema.
2. Identificar cada componente a nivel abstracto y sus funciones.
3. Diseñar una solución con dichos componentes.

La implementación es una etapa posterior...

EL DISEÑO

- Proceso:
 - Elaborar un plan de lo que se quiere hacer
(concretar el detalle interno irá en un paso posterior)
- Resultado:
 - Producto de ingeniería de calidad (de ingeniería informática), es decir: software de calidad

PROGRAMACIÓN CON ABSTRACCIÓN (TIPOS ABSTRACTOS DE DATOS):

- Hay una etapa previa de **DISEÑO**

TAD / CLASE en POO

TAD Dados

Representa el lanzamiento de dos dados.

OPERACIONES

- Lanzamiento: simula el lanzamiento de 2 dados.
- GetDado1: devuelve el valor del primer dado.
- GetDado2: devuelve el valor del segundo dado.
- GetSuma: devuelve la suma de los dos dados.

Este texto **destaca** lo que queremos hacer y **oculta** los detalles internos. Se llama: **ESPECIFICACIÓN**
(porque debe 'especificar bien' lo que se pretende y debe ser un texto preciso)

Class Dados

DISEÑO

dados.h

```
class Dados{  
    private:  
        int d1_, d2_;  
    public:  
        void Lanzamiento();  
        int GetDado1();  
        int GetDado2();  
        int GetSuma();  
        . . .  
};
```

juego.cc

```
Dados d;  
d.Lanzamiento();  
cout << "d1 = " << d.GetDado1();  
cout << "d2 = " << d.GetDado2();  
cout << "sum = " << d.GetSuma();
```

Comparando...

juego.c

```
int d1, d2, sum, r1, r2;
srand(seed);
r1 = rand();
r2 = rand();
d1 = (r1 % 6) + 1;
d2 = (r1 % 6) + 1;
sum = d1 + d2;
printf("d1 = %d\n", d1);
printf("d2 = %d\n", d2);
printf("sum = %d\n", total);
```

juego.cc

```
Dados d;
d.Lanzamiento();
cout << "d1 = " << d.GetDado1();
cout << "d2 = " << d.GetDado2();
cout << "sum = " << d.GetSuma();
```

dados.cc

IMPLEMENTACIÓN

```
Dados::Datos()  
{  
    srand(time(NULL));  
}  
void Dados::Lanzamiento()  
{  
    d1_=(rand()%6)+1;  
    d2_=(rand()%6)+1;  
}
```

```
int Dados::GetDado1()  
{  
    return d1_;  
}  
int Dados::GetDado2()  
{  
    return d2_;  
}  
int Dados::GetSuma()  
{  
    return d1_+d2_;  
}
```

ABSTRACTION. EXAMPLE B

- Necesito enviar un mensaje a una Red Social

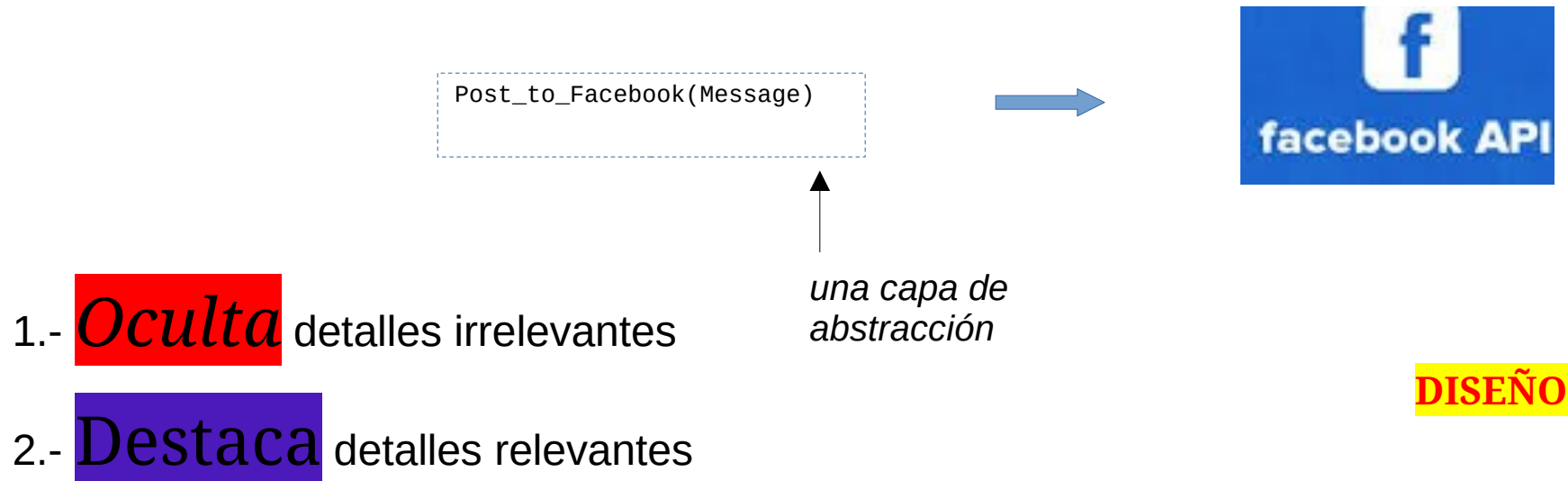
```
// for Facebook verification
app.get('/webhook/', function (req, res) {
  if (req.query['hub.verify_token'] === 'my_voice_is_my_password_verify_me') {
    res.send(req.query['hub.challenge'])
  }
  res.send('Error, wrong token')
})
// Spin up the server
app.listen(app.get('port'), function() {
  console.log('running on port', app.get('port'))
})
```



Puede hacerse así, pero esta solución le falta algo, debo dotarlo co

ABSTRACTION. EXAMPLE B

- Necesito enviar un mensaje a una Red Social



ABSTRACTION. EXAMPLE B

me estoy
repitiendo?

Post_to_Facebook(Message)



Post_to_Twitter(Message)

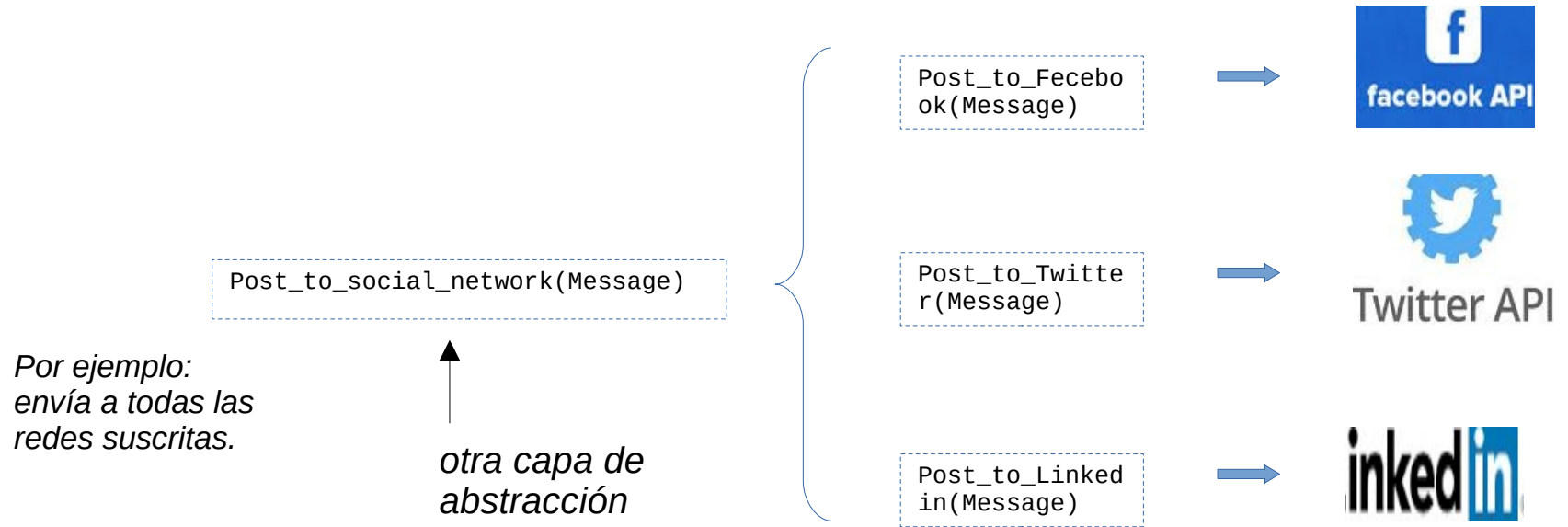


Post_to_Linkedin(Message)



DISEÑO

ABSTRACTION. EXAMPLE B



1.- **Ocultar** detalles irrelevantes

2.- **Destacar** detalles relevantes

*una capa de
abstracción*

DISEÑO

Cliente o usuario de una clase

```
#include "book.h"
int main(void)
{
    Book b;
    . . .
    b.getTitle();
    . . .
}
```

cliente/usuario

Este programa es **cliente** de la clase Book

El autor de este programa (el cliente):

- No tiene por qué ser el mismo que el autor de la clase Book.
- Desconoce los datos internos de la clase Book (tampoco los necesita...).
- Desconoce cómo funciona por dentro la clase Book (desconoce cómo está implementada internamente).
- Ni siquiera debe preocuparle como esté hecha por dentro la clase Book.
- Solo debe preocuparle el programa que está haciendo en ese momento.
- Se conecta a cada objeto a través de su interfaz o parte “**public**” de la clase.
- No puede acceder a la parte “**private**” de la clase.

Encapsulamiento

- Una clase tendrá unos datos internos inaccesibles desde su API.
- Los datos internos irán en la **sección “private”** y no podrá accederse a ellos directamente.
- Las operaciones (el API) irán en la **sección “public”** y serán el único método para acceder al objeto.
- Se dice que los datos internos quedan ocultos, encapsulados en la clase.
- Los datos internos de un objeto también se denomina “estado de un objeto”.

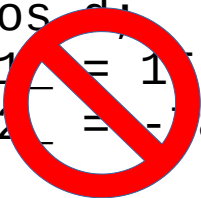
Encapsulamiento. Ventajas

- Impide acceso directo al estado interno de un objeto:
 - Impidiendo operaciones no permitidas.
 - Simplifica su comprensión ya que no será necesario conocer como está hecho internamente.
 - Si en el futuro se modifican sus datos internos, no afectará a ningún programa que use dicho objeto ya que nunca se accede a ellos.
- Solo podrá interactuarse con el objeto mediante su **interfaz** pública, su API.

Encapsulamiento. Ejemplos

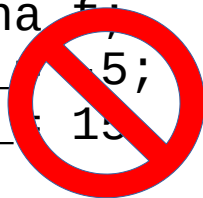
```
class Datos{  
private:  
    int d1_  
    int d2_  
    . . .  
public:  
    . . .  
};
```

Datos d;
d.d1_ = 1;
d.d2_ = 8



```
class Fecha{  
private:  
    int d_  
    int m_  
    int a_  
    . . .  
public:  
    . . .  
};
```

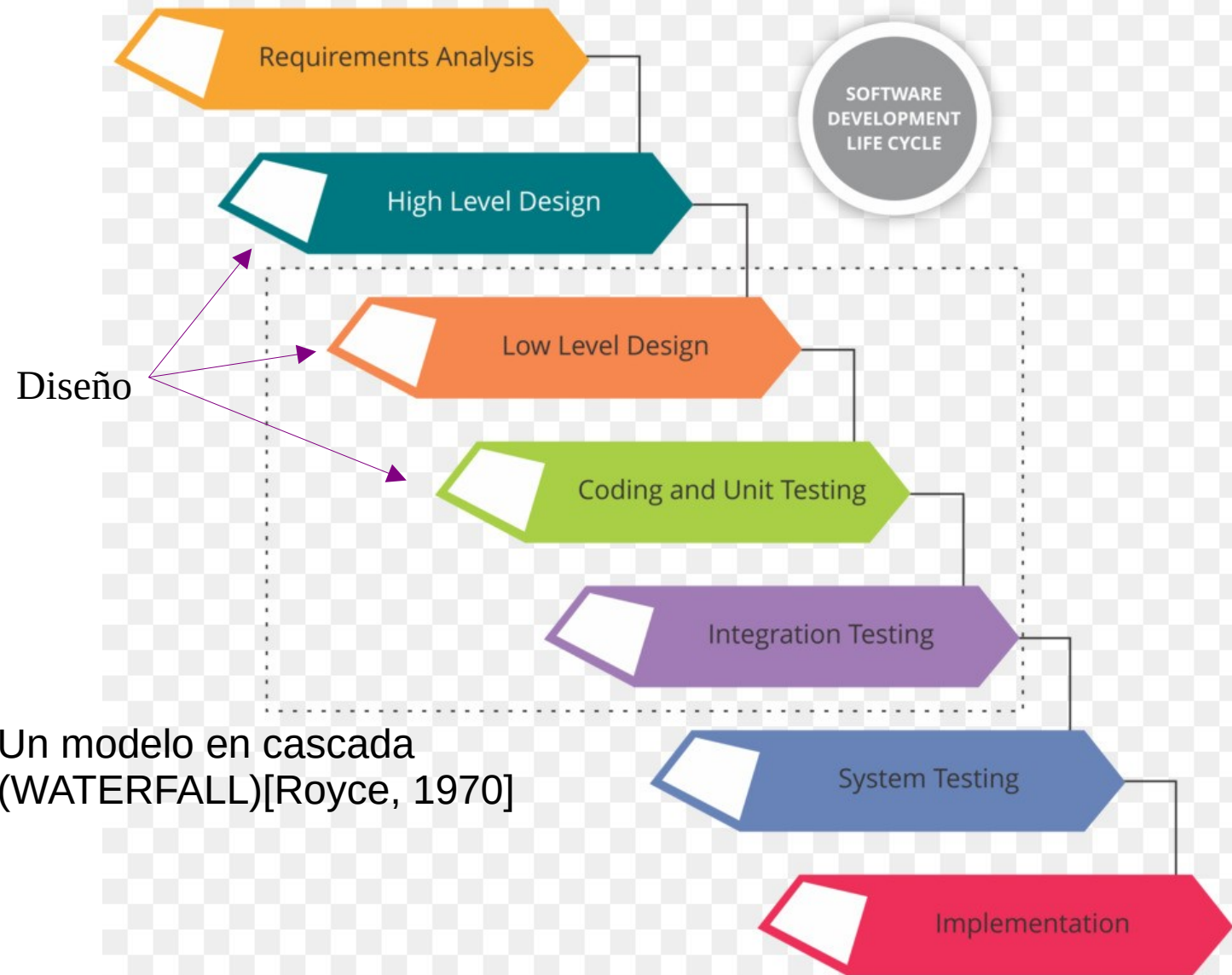
Fecha f;
f.d_ = 5;
f.m_ = 15



Son operaciones no permitidas, estados erróneos del objeto: NO SE DEBEN PERMITIR
Los objetos son protegidos mediante el **ENCAPSULAMIENTO**

Claves del Diseño Orientado a Objetos

- ¿Quién y qué?
- Dejamos a un lado el ¿cómo?
- ¿Quién?: Las clases (herencia, polimorfismo, sobrecarga)
- ¿Qué?: Los métodos de las clases, la interfaz
- También diseñamos: las pruebas
- Diseñando utilizamos: patrones de diseño



TEMA 2

SOFTWARE DE CALIDAD

Software de calidad

- Modelo de McCall
- Calidad funcional (externa) y estructural (interna) del software
- Factores de calidad:
 - Factores de calidad externos
 - Factores de calidad internos

Factores de calidad del software

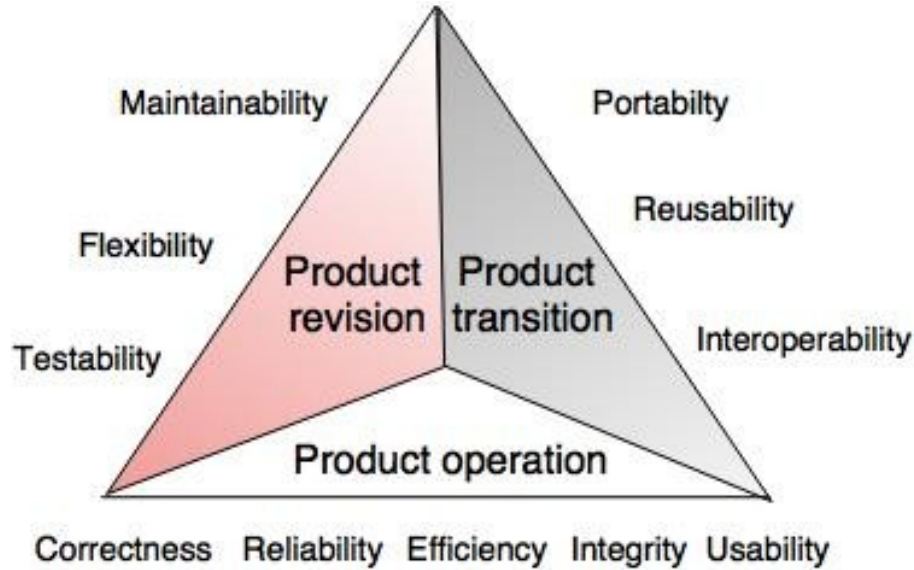


Fig. - McCall's quality factors

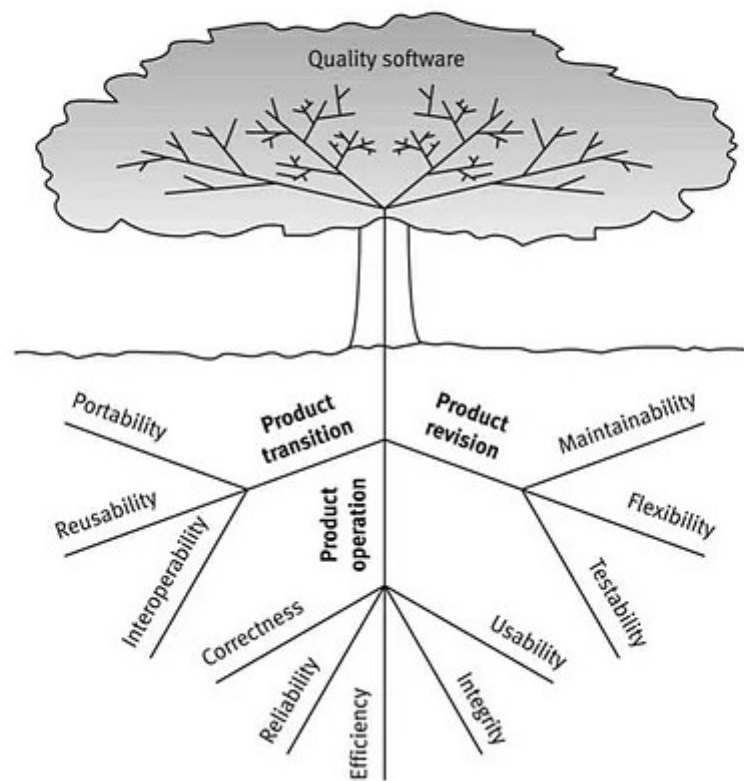
[McCall y colaboradores, 1977]

Product operation:
relacionado con la
funcionalidad

Product revision:
mantenimiento

Product transition:
adaptación a otros
entornos e interrelación
con otro software

McCalls factor model tree



Factores de calidad del software

| Quality Categories | Quality Factors | Broad Objectives |
|--------------------|--|---|
| Product operation | Correctness Reliability Efficiency Integrity Usability | Does it do what the customer wants? Does it do it accurately all of the time? Does it quickly solve the intended problem? Is it secure? Can I run it? |
| Product revision | Maintainability Testability Flexibility | Can it be fixed? Can it be tested? Can it be changed? |
| Product transition | Portability Reusability Interoperability | Can it be used on another machine? Can parts of it be reused? Can it interface with another system? |

(Estudiar las definiciones en los apuntes del profesor)

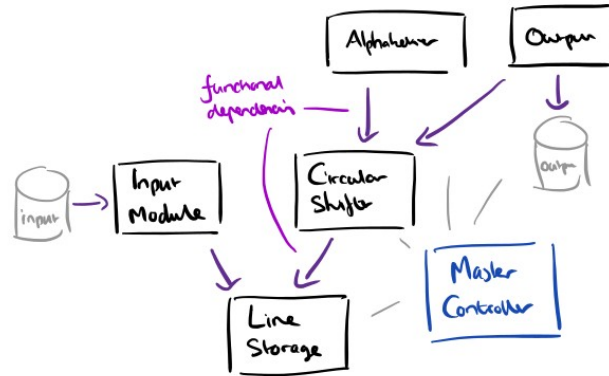
Otros Factores de calidad

- Seguridad
- Accesibilidad
- Oportunidad
- Economía

... siempre debe existir un compromiso
entre factores de calidad

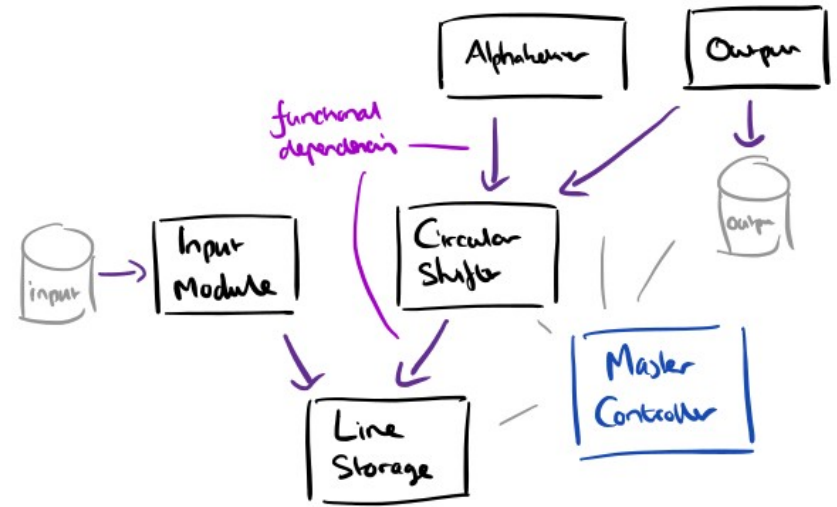
TEMA 3

DESCOMPOSICIÓN MODULAR



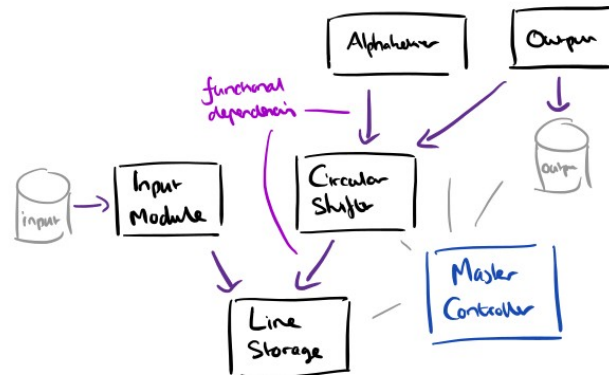
Descomposición modular

- ¿Cómo se hace?
- ... ¿Los archivos de código fuente?
¿Las funciones?...
- En POO hablamos de clases,
pero: ¿cómo se hace una
buena descomposición modular?



Diseñar una buena descomposición

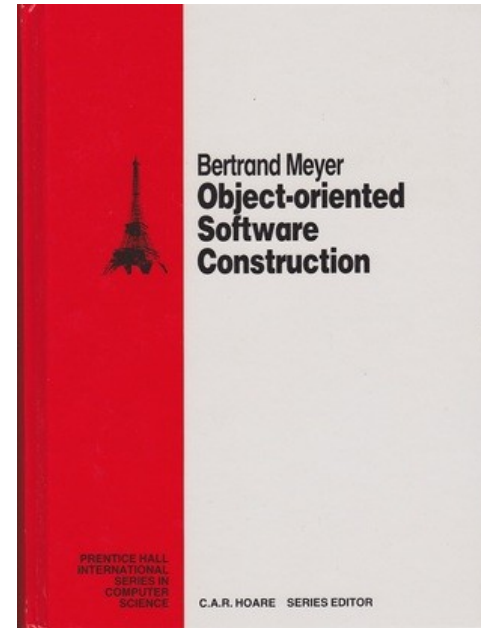
- Frente a un mismo problema pueden darse diferentes estructuras de descomposición
- ¿Cuál de ellas es mejor que las otras?
- ¿Por qué?



Descomposición Modular

Bertrand Meyer (Francia, 1950). *Construcción de Software Orientado a Objetos*. 1999:

- Criterios/requisitos
- Reglas
- Principios



Descomposición 1/3. Criterios

- 1) Descomposición modular
- 2) Composición modular
- 3) Comprensibilidad modular
- 4) Continuidad modular
- 5) Protección modular

Descomposición 2/3. Reglas

- 1) Correspondencia directa
- 2) Pocas interfaces
- 3) Pequeñas interfaces
- 4) Interfaces explícitas
- 5) Ocultación de la información

Descomposición 3/3. Principios

- 1) Unidades modulares lingüísticas
- 2) Auto-documentación
- 3) Acceso uniforme
- 4) Principio abierto-cerrado
- 5) Elección única

Referencias / Bibliografía

- Apuntes del profesor “Calidad del Software y Factores de Calidad”
- Apuntes del profesor “Descomposición Modular”
- Cavano, J.P., McCall, J.A., A Framework for the Measurement of Software Quality, Proc. of the ACM Software Quality Assurance Workshop, pp. 133139, Nov. 1978.
- Bertrand Meyer. Object-Oriented Software Construction. 1999.

Fin Tema 3