Ingeniería Software para Sistemas Empotrados

# Introducción al diseño software

Carlos Carreras

Departamento de Ingeniería Electrónica
Universidad Politécnica de Madrid

1

## Contenidos

- Métodos de diseño software. Criterios de evaluación
- Diseño funcional frente a diseño orientado a objetos
- Aspectos clave en el diseño software. Ejemplos
- Reglas de diseño
- El lenguaje C++. Objetivos y características principales

- *Agradecimientos: Douglas C. Schmidt (Vanderbilt University)*

2

## Contenidos

- **Métodos de diseño software. Criterios de evaluación**
- Diseño funcional frente a diseño orientado a objetos
- Aspectos clave en el diseño software. Ejemplos
- Reglas de diseño
- El lenguaje C++. Objetivos y características principales

3

## Diseño algorítmico o funcional

- Diseño *top-down* basado en las funciones que realiza el sistema
- Normalmente sigue una estrategia “divide y vencerás” basada en funciones
  - Las funciones más generales se descomponen en otras más específicas de forma iterativa/recursiva
- Los componentes de diseño primarios corresponden a pasos de procesamiento en la secuencia de ejecución
  - Es similar a una receta de cocina

4

## Diseño orientado a objetos

- Diseño basado en modelar clases y objetos en el dominio de aplicación
  - Pueden reflejar o no el “mundo real”
- Normalmente sigue una estrategia de “abstracción de datos jerárquica” donde los componentes se basan en clases, objetos, módulos y procesos
- Las operaciones se relacionan con objetos o clases de objetos específicos
- Grupos de clases y de objetos se combinan a menudo como *frameworks*

5

## Diseño estructurado

- Diseño basado en las estructuras de datos de entrada y salida del sistema
- Normalmente sigue una estrategia de descomposición basada en el flujo de datos entre componentes de procesamiento (base de sistemas de procesado de datos)
- El diseño depende del ordenamiento temporal de las fases de procesamiento
- Los cambios en la representación de datos afectan a toda la estructura al no ocultarse los datos

6

## Sistemas transformacionales

- Diseño basado en la especificación del problema, no en la especificación de la solución
  - La solución se deriva automáticamente de la especificación de alto nivel
- Cada componente transformacional puede ser implementado siguiendo otras alternativas de diseño
- Su uso está limitado a dominios bien conocidos:
  - Generadores de traductores, constructores de interfaces gráficas de usuario, procesado de señal

7

## Criterios para evaluar métodos de diseño

- Descomposición
  - ¿Ayuda a la descomposición en subproblemas?
- Composición
  - ¿Permite crear nuevos sistemas con componentes existentes?
- Comprensión
  - ¿Se pueden entender los componentes por separado?
- Continuidad
  - ¿Es limitado el efecto de pequeños cambios de especificación?
- Protección
  - ¿Está limitado el efecto de anormalidades en la ejecución?
- Compatibilidad
  - ¿Es uniforme la definición de interfaces entre componentes?

8

## Contenidos

- Métodos de diseño software. Criterios de evaluación
- **Diseño funcional frente a diseño orientado a objetos**
- Aspectos clave en el diseño software. Ejemplos
- Reglas de diseño
- El lenguaje C++. Objetivos y características principales



## Caso de estudio: verificador ortográfico (*spell checker*)

- Descripción del sistema:
  - "Revisa las palabras de un documento concreto, buscándolas en un diccionario (general y/o definido por el usuario) compuesto de palabras. Muestra en la salida estándar (pantalla de terminal) aquellas palabras que no aparezcan en algún diccionario ni puedan ser derivadas de aquellas que aparecen mediante la aplicación de ciertas inflexiones, prefijos o sufijos."
- Descripción algorítmica en pseudo-código:
  1. Obtener el nombre del fichero del documento
  2. Extraer lista de palabras, ordenarlas y descartar repeticiones
  3. Buscar cada palabra en el (o los) diccionario(s)
     a. Si está o se puede derivar según distintas reglas, ignorarla
     b. Si no, mostrarla en salida estándar como palabra incorrecta

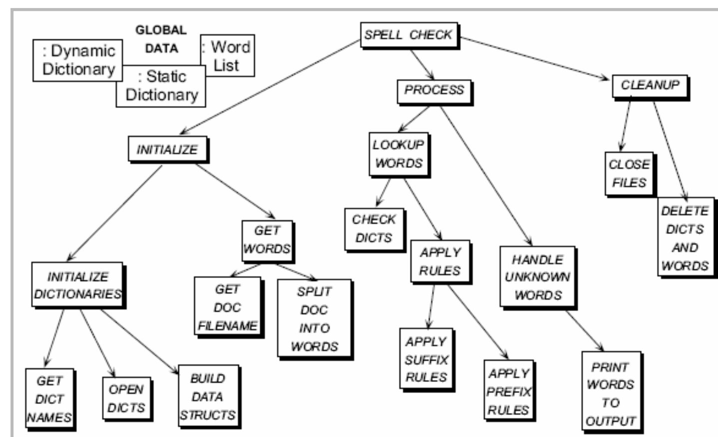
## Diseño algorítmico

- Organización basada en las funciones del sistema
  - El refinamiento de funciones precede y guía al de datos
- Refinamiento top-down de funciones:
  - Partir la función del sistema en subfunciones
  - Determinar el flujo de datos entre ellas (definir estructuras de datos)
  - Iterar hasta que la implementación sea inmediata y obvia
- Preguntas importantes:
  - ¿Cómo afectan al diseño cambios posteriores en la especificación y/o la implementación?
  - ¿Son reutilizables los componentes desarrollados?

11

## Diseño algorítmico: estructura (*structure chart*)

- Expresa la jerarquía y el flujo de datos




12

## Diseño algorítmico: programa

```
struct List {
  char *word;
  struct List *next, *prev;
} *list = 0;
extern struct Static_Dictionary main_dict;
extern struct Dynamic_Dictionary private_dict;

int main (int argc, char *argv[]) {
  initialize (); /* Perform initializations */
  process (); /* Perform lookups */
  cleanup (); /* Deallocate resources */
}

int process (void) {
  for (struct List *ptr = list;
       ptr != 0;
       ptr = ptr->next)
    if (static_lookup (&main_dict, ptr->word) ||
        dynamic_lookup (&private_dict, ptr->word))
      {
        struct List *tmp = ptr;
        ptr->prev->next = ptr->next;
        free (tmp);
      }
  handle_unknown_words ();
}
```

13

## Diseño algorítmico: conclusiones

- Ventajas:
  - Apropiado para pequeños programas con algoritmos intensivos: Ocho Reinas, Torres de Hanoi, ordenar, buscar, etc.
  - Fácil de entender en problemas pequeños (estructura “verbal”)
  - Intuitivo para muchos diseñadores (coincide con su formación)
- Desventajas:
  - No facilita la evolución del sistema (pequeños cambios afectan a todo el programa, abundancia de variables globales, ...)
  - No facilita la reutilización (responde a una aplicación concreta)
  - La importancia de las estructuras de datos está infravalorada

14

# Diseño orientado a objetos

- Organización basada en descomponer el sistema en clases y objetos
  - La implementación de lo que puede cambiar se oculta
  - El orden de actividades se considera más tarde, lo que da una idea de subespecificación que es intencionada pues se pretende desarrollar componentes reutilizables
- Reto fundamental: identificar objetos y clases
  - Posibilidad: usar palabras o frases de la especificación para
    - Identificar objetos, operaciones y atributos
    - Determinar las interacciones y la visibilidad

15

# Diseño orientado a objetos: identificar clases y objetos

- Ejemplo: verificador ortográfico
  - "*Revisa* las **palabras** de un **documento** concreto, *buscándolas* en un **diccionario** (general y/o definido por el usuario) *compuesto* de **palabras.** *Muestra* en la **salida** estándar (pantalla de terminal) aquellas **palabras** que no aparezcan en algún **diccionario** ni puedan *ser derivadas* de aquellas que aparecen mediante la *aplicación* de ciertas **inflexiones**, **prefijos** o **sufijos.**"
  - **Nombres comunes**: clases
  - **Nombres** propios: objetos (instancias de clases)
  - *Verbos*: acciones en objetos
  - **Otros:** Adverbios: restricciones en ops (inserción rápida => no verificar rango)
    Adjetivos: atributos (gran diccionario => campo de tamaño)
    Relación sujeto-objeto: dependencias (diccionario de palabras)

16

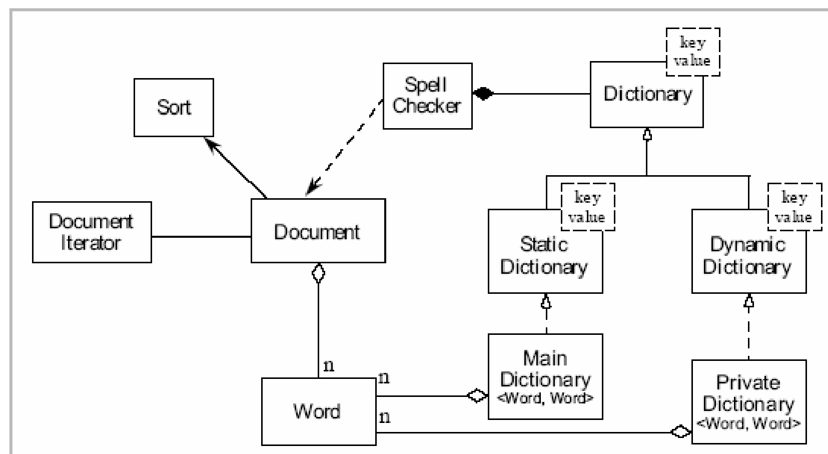
## Diseño orientado a objetos: aplicación

- La visibilidad sólo debe satisfacer las dependencias
  - En general: reducir la visibilidad global, desacoplar, y poner énfasis en la cohesión
  - Ejemplo: **Documento** y **Diccionario** no deberían ser visibles fuera del contexto **Verificador ortográfico**

- Desarrollar un conjunto de diagramas que describan gráficamente las relaciones entre clases, objetos, módulos y procesos desde distintas perspectivas

17

## Diseño orientado a objetos: diagrama de clases




18

# Diseño orientado a objetos: otros diagramas

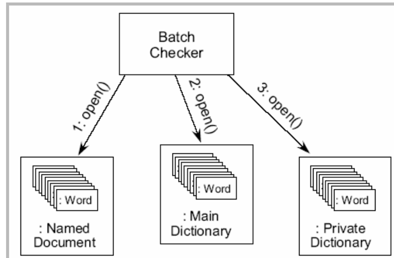



Diagrama de objetos

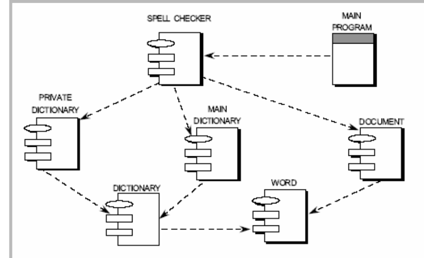
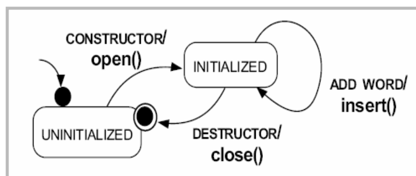



Diagrama de módulos




Máquina de estados de **Diccionario**

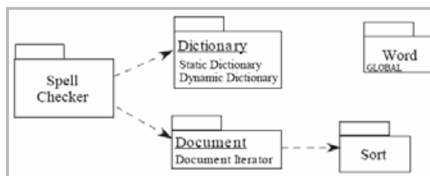



Diagrama del paquete

19

# Diseño orientado a objetos: descripción de clases como bloques

```
NAME          Word
ACCESS        Exported
CARDINALITY   Unlimited
MEMBERS       construct/destruct
              insert/remove characters
              clone
              concatenate
              compare
              ...

NAME          Document
ACCESS        Exported
CARDINALITY   Unlimited
MEMBERS       construct/destruct
              next word iterator
              sort
              ...

NAME          Spell_Checker
ACCESS        Exported
CARDINALITY   Unlimited
MEMBERS       construct/destruct
              spell_check
              ...
```

```
NAME            Dictionary
QUALIFICATIONS  Abstract class
ACCESS          Exported
CARDINALITY     Unlimited
MEMBERS         construct/destruct
     open/close
                insert word
                find word
                remove word
                next word iterator
                ...

NAME            Dynamic Dictionary
ACCESS          Exported
CARDINALITY     Unlimited
SUPERCLASS      Dictionary
MEMBERS         construct/destruct
                ...

NAME            Static Dictionary
ACCESS          Exported
CARDINALITY     Unlimited
SUPERCLASS      Dictionary
MEMBERS         construct/destruct
                ...
```

20

# Diseño orientado a objetos: implementación de clases (1/3)

```
class Word {
public:
  Word (void);
  Word (const string &);
  int insert (int index, char c);
  int clone (Word &);
  int concat (const Word &);
  int compare (const Word &);
  // ...
};

class Document {
public:
  Document (void);
  ~Document (void);
  virtual int open (const string &filename);
  int sort (int options);
  // ...
};

class Document_Iterator {
public:
  Document_Iterator (const Document &);
  int next_item (Word &);
  // ...
};
```

```
namespace Dictionary {
template <class KEY, class VALUE>
class Dictionary {
public:
  virtual int open (const string &filename) = 0;
  virtual find (KEY, VALUE &) = 0;
  virtual insert (KEY, VALUE &) = 0;
  virtual remove (KEY) = 0;
  // ...
};

template <class KEY, class VALUE>
class Dynamic_Dictionary : public Dictionary {
public:
  virtual int open (const string &filename);
  virtual find (KEY, VALUE &);
  // ...
};

template <class KEY, class VALUE>
class Static_Dictionary : public Dictionary {
public:
  virtual int open (const string &filename);
  virtual find (KEY, VALUE &);
  // ...
};
}
```

21

# Diseño orientado a objetos: implementación de clases (2/3)

```
#include "Document.h"
#include "Static_Dictionary.h"
#include "Dynamic_Dictionary.h"

using namespace Dictionary;
typedef Static_Dictionary<Word, Word> Main_Dictionary;
typedef Dynamic_Dictionary<Word, Word> Private_Dictionary;

class Spell_Checker {
public:
  ~Spell_Checker (void);
  int open (const string &doc_name,
            const string &main_dict_name,
            const string &private_dict_name);
  int spell_check (ostream &standard_output);
private:
  Document named_document;
  Main_Dictionary main_dictionary;
  Private_Dictionary private_dictionary;
};
```

22

# Diseño orientado a objetos: implementación de clases (3/3)

```
Spell_Checker::Spell_Checker (const string &doc_name,
                              const string &main_dict_name,
                              const string &private_dict_name)
{
  if (named_document.open (doc_name) == -1
      || main_dictionary.open (main_dict_name) == -1
      || private_dictionary.open (private_dict_name) == -1) {
       cerr << "intialization problem";
       throw Invalid_Name ();
  }
}
```

```
int Spell_Checker::spell_check (ostream &standard_output)
{
  int result = 0;
  Word word;
  named_document.sort (REMOVE_DUPS);
  for (Document_Iterator doc_iter (named_document);
       doc_iter.next_item (word) != -1; )
    if (main_dictionary.find (word) != -1
        || private_dictionary.find (word) != -1)
      continue; // found word
    else {
      standard_output.write (word);
      // erroneous word
      result = -1;
    }
  // ...
```

23

# Diseño orientado a objetos: programa principal

```
int main (int argc, char *argv[])
{
  if (argc != 4) {
    cerr << "usage: " << argv[0]
         << ": doc-name, main-dict, private-dict"
         << endl;
    return 1;
  }
  Spell_Checker batch_checker (argv[1], argv[2],
                               argv[3]);
  if (batch_checker.spell_check (cout) == -1)
    return -1;
  else
    return 0;
}
```

- Aunque la descomposición orientada a objetos usa básicamente el mismo algoritmo que el diseño algorítmico, la arquitectura software es totalmente diferente

24

## Diseño orientado a objetos: conclusiones

- Ventajas:
  - Modularidad
    - Fácil comprensión: se reduce el acoplamiento y la visibilidad
    - Adaptabilidad a cambios: las interfaces suelen mantenerse y los detalles de representación son internos a las clases y están ocultos al resto
  - Igual énfasis en datos y funciones dentro de las clases
  - Comportamiento de objetos independiente de ordenación temporal (facilita la reutilización y la ampliación)
- Desventajas:
  - Más trabajo (análisis, modelado,...) y menos intuitivo
  - En algunos casos no aporta ventajas (rutinas matemáticas)



## Contenidos

- Métodos de diseño software. Criterios de evaluación
- Diseño funcional frente a diseño orientado a objetos
- Aspectos clave en el diseño software. Ejemplos
- Reglas de diseño
- El lenguaje C++. Objetivos y características principales

## Objetivos de la fase de diseño

- Descomponer el sistema en componentes (identificar la arquitectura del sistema)
- Determinar las relaciones entre componentes (identificar las dependencias)
- Definir los mecanismos de comunicación (memoria compartida, llamadas a función, RPC, etc.)
- Especificar las interfaces de los componentes (facilita el test y la comunicación entre programadores)
- Describir la funcionalidad de los componentes
- Identificar oportunidades para una reutilización sistemática

27

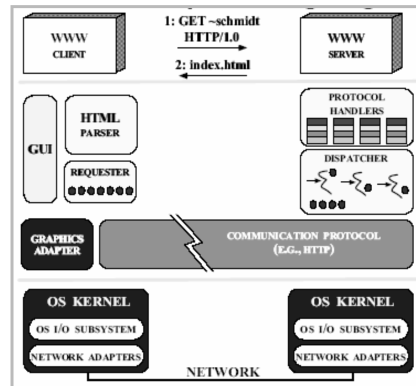
## El proceso de diseño

- Diseñar es un proceso de decisión iterativo:
  1. Listar las decisiones difíciles y las que puedan cambiar
  2. Hacer una especificación de componentes que oculte cada una de esas decisiones
     - Considerar primero las decisiones que afecten a la familia de programas
     - Modularizar primero los cambios más probables
     - Simultáneamente, diseñar la jerarquía de reutilización
  3. Tratar cada componente como una especificación a la que se aplican los dos pasos anteriores
  4. Continuar hasta que cada decisión de diseño esté oculta en un componente fácilmente comprensible y que permita una implementación de bajo nivel individual e independiente

28

# Ejemplo de diseño: servidor web

- Decisiones de diseño:
  - Aspectos de portabilidad
  - Demux y concurrencia de E/S
  - Procesado del protocolo HTTP
  - Acceso a ficheros
- Componentes:
  - Gestor de eventos
  - Manejador del protocolo
  - Sistema de ficheros virtual




29

# Conceptos clave y principios

- Conceptos clave y principios del diseño software:
  1. Descomposición
  2. Abstracción y ocultación de la información
  3. Modularidad mediante componentes
  4. Ampliación
  5. Arquitecturas de máquinas virtuales
  6. Relaciones jerárquicas
  7. Familias de programas y subconjuntos
- Objetivos que se persiguen:
  - Manejar la complejidad del sistema software
  - Mejorar los factores de calidad del software
  - Facilitar una reutilización sistemática
  - Resolver retos de diseño habituales

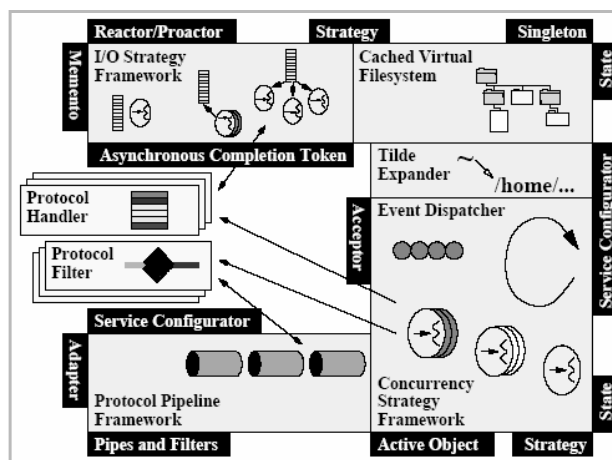
30

# Reto 1: determinar la arquitectura del servidor web (1/2)

- Objetivos: evitar un diseño monolítico, facilitar el trabajo concurrente, la portabilidad y la reutilización
- Solución: descomposición según cierto paradigma (p.e. orientada a objetos)
- Metodología (iterativa):
  - Seleccionar una parte del problema (inicialmente todo)
  - Determinar los componentes (diseño orientado a objetos)
  - Describir las interacciones entre componentes

31

# Reto 1: determinar la arquitectura del servidor web (2/2)




http://cs.wustl.edu/~schmidt/PDF/JAWS.pdf

32

## Reto 2: implementación flexible del servidor web

- Objetivo: evitar componentes fuertemente acoplados para que el sistema pueda evolucionar (nuevas plataformas, compiladores, funcionalidades, prestaciones)
- Solución: Abstracción y ocultación de la información tras interfaces abstractas bien definidas
- Tipos de abstracción:
  - Abstracción procedural (p.e. subrutinas frente a código insertado)
  - Abstracción de datos (p.e. tipos de datos abstractos ADT)
  - Abstracción de control (p.e. bucles, iteradores, etc.)

33

## Ocultación de la información

- Interfaces abstractas
  - Permiten ocultar información que podría cambiar
  - Deben ser el medio de comunicación del software de aplicación
  - Deben especificarse con la mínima información posible
- Información que debe ocultarse habitualmente
  - En representación de los datos: uso de tipos abstractos
  - En algoritmos: técnicas utilizadas (para ordenar, buscar, ...)
  - En formatos de E/S: dependencia de máquinas, orden de bits, ...
  - En interfaces de bajo nivel: orden de operaciones
  - En general, separar políticas de mecanismos

34

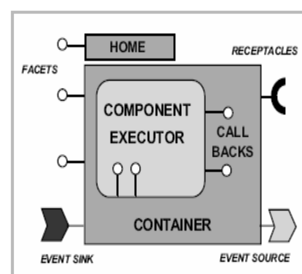
## Reto 3: unidades de descomposición del servidor web

- Objetivo: identificar las unidades para la descomposición que permiten interfaces abstractas y al mismo tiempo mantienen alta cohesión y bajo acoplamiento
- Solución: modularidad mediante componentes
- Componente: unidad de código que
  - Tiene uno o más nombres
  - Tiene límites conocidos
  - Es (re-)usable por otros componentes
  - Encapsula datos
  - Oculta detalles innecesarios
  - Es compilable por separado



## Diseño de interfaces de componentes

- Tipos de puertos en interfaces:
  - Exports: servicios ofrecidos a otros componentes (facetas y fuentes de eventos)
  - Imports: servicios solicitados a otros componentes (receptáculos y sumideros de eventos)
  - Control de acceso: diferenciación entre clientes (p.e. privado, protegido, público)




- Es aconsejable definir componentes con varias interfaces e implementaciones, anticipando los cambios

## Beneficios de la modularidad

- Facilita mejoras en factores de calidad software
  - Extensibilidad, gracias a las interfaces abstractas
  - Reutilización, gracias al bajo acoplamiento y la alta cohesión
  - Compatibilidad, gracias al diseño de interfaces "puente"
  - Portabilidad, al ocultar las dependencias de máquinas
- Además, es importante para conseguir buenos diseños al
  - Enfatizar la separación de ámbitos
  - Reducir la complejidad de diseño gracias a arquitecturas descentralizadas
  - Basarse en un desarrollo concurrente e independiente que incrementa la escalabilidad

37

## Diseños modulares

- Un diseño es modular si permite:
  - Descomposición de componentes grandes en otros menores
  - Composición de componentes grandes con otros menores
  - Comprensión de componentes por separado
  - Propagación limitada de cambios en componentes (continuidad)
  - Propagación limitada de anormalidades en ejecución
- Principios que aseguran diseños modulares
  - Que el lenguaje permita componentes como unidades sintácticas
  - Que se usen pocas interfaces (pequeñas) para la comunicación
  - Que la comunicación sea explícita y de poca información
  - Que la información de un componente sea privada por defecto

38

## Reto 4: desarrollo para cambios futuros del servidor web

- Objetivo: facilitar la inclusión de cambios que sólo se revelan como necesarios cuando el sistema está acabado
- Solución: ampliación del sistema
- Componente ampliable: sigue el principio abierto/cerrado
  - Abierto: debe permitir ampliaciones, pues siempre hay posibles nuevas mejoras, oportunidades de mercado, etc.
  - Cerrado: debe poder ser usado por otros componentes sin que sus cambios impliquen cambios en esos otros componentes

39

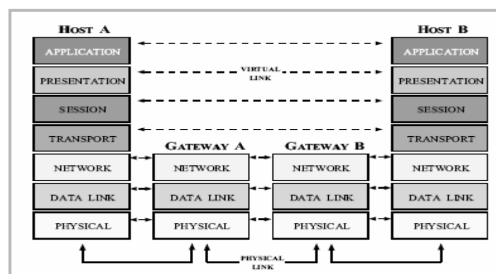
## Reto 5: separar ámbitos en un servidor web con múltiples capas

- Objetivo: permitir cambios futuros en sistemas basados en multiples capas (*layers*) con las que se facilita
  - La reutilización de servicios de capas inferiores por otros de capas superiores
  - La ampliación de la funcionalidad de un modo incremental y transparente
  - La mejora de prestaciones al evitar selectivamente funcionalidad innecesaria
  - La mejora de las implementaciones, el test y el manteniemiento
- Solución: arquitecturas de máquinas virtuales

40

## Arquitecturas de máquinas virtuales

- Una máquina virtual provee un "conjunto de instrucciones software" ampliado (nuevas operaciones en cada capa)
  - Las ampliaciones proporcionan nuevos tipos de datos y sus correspondientes "instrucciones software"
  - Se modelan a partir del conjunto de instrucciones hardware que funciona con un número limitado de tipos de datos




41

## Ejemplos de máquinas virtuales

- El toolkit ACE (Adaptive Communication Environment)
  - www.cs.wustl.edu/~schmidt/ACE.html
  - Capa de SO: procesos, comunicaciones, memoria virtual, ficheros
  - Capa de APIs: procesos/hilos, sockets, fifos, memoria compartida
  - Capa de frontal C++: gestor de procesos, sincronismo, malloc
  - Capa de framework: manejador de servicios, de corba, de conexión
  - Capa de servicios de red: servidor de acceso, de nombres, de web
- Otras máquinas virtuales (capas)
  - Arquitecturas de ordenadores: compilador, ensamblador, código objeto, microcódigo, puertas, transistores
  - Sistemas operativos: conjunto de instrucciones, llamadas al sistema, interrupciones, su manejador, sus máscaras, su stack
  - Máquina virtual JAVA: abstrae los detalles del SO subyacente

42

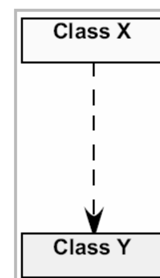
## Reto 6: separar ámbitos en un servidor web jerárquico

- Objetivo: permitir cambios futuros en sistemas jerárquicos que manejan diferentes niveles de abstracción
- Solución: relaciones jerárquicas
- Jerarquías:
  - Reducen la interacción entre componentes al limitar la topología de sus relaciones
  - Una relación define una jerarquía si parte las unidades en niveles
    - Nivel 0: unidades que no usan otras unidades
    - Nivel i: unidades que usan al menos una de nivel $<i$ y ninguna de nivel $\geq i$
  - Forman la base de las arquitecturas y de los diseños: desarrollo independiente, aislan los cambios, permiten prototipado rápido

43

## Definición de jerarquías: la relación *Usa*

- X *Usa* Y si el correcto funcionamiento de X depende de que esté disponible una implementación correcta de Y
  - No es necesariamente lo mismo que *Invoca*
    - Un listado de errores invoca sin usar (*error logging*)
    - Un paso de mensajes usa sin invocar (*message passing*)
  - No define necesariamente una jerarquía
  - Es esencial para definir sistemas reutilizables

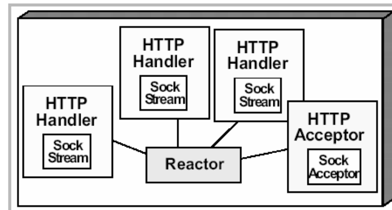



- Permitir que X use Y cuando:
  - X es más sencilla por usar Y (p.e. clases stdlib C++)
  - Y no es mucho más compleja al no poder usar X
  - Hay un subconjunto útil que contiene Y y no X
  - No hay ningún subconjunto útil con X y no Y

44

## Definición de jerarquías: la relación *Se-Compone-De*

- X *Se-Compone-De* $\{x_i\}$ si X es un grupo de componentes $x_i$ que comparten un propósito común
  - Muestra cómo el sistema se compone de componentes



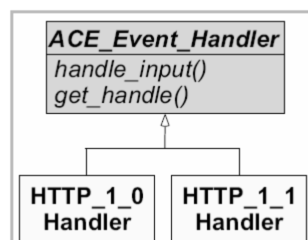

Algunas relaciones *Se-Compone-De* en el sistema JAWS

- Los siguientes conceptos no describen lo mismo:
  - Nivel o capa en una máquina virtual
  - Componente, que es una entidad que oculta información
  - Subprograma, que es una unidad de código

45

## Definición de jerarquías: la relación *Es-Un*

- La clase X tiene una relación *Es-Un* con la clase Y si las instancias de la clase X son una especialización de Y
  - Es una relación orientada a objetos (se habla de clases)
  - Es una relación "antepasado/descendiente" asociada con las características de herencia del lenguaje

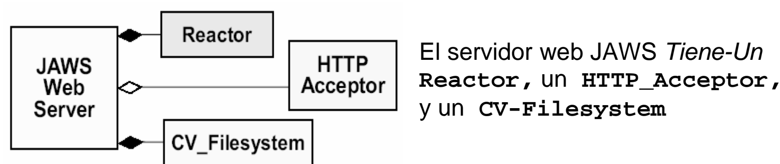



Un `HTTP_1_0_Handler` *Es-Un* `ACE_Event_Handler` que se especializa en procesar solicitudes HTTP 1.0

46

## Definición de jerarquías: la relación *Tiene-Un*

- La clase X tiene una relación *Tiene-Un* con la clase Y si las instancias de la clase X tienen una instancia de la clase Y
  - Es una relación orientada a objetos (se habla de clases)
  - Es una relación “cliente” asociada con lenguajes que manejan clases y objetos




El servidor web JAWS *Tiene-Un* `Reactor`, un `HTTP_Acceptor`, y un `CV-Filesystem`

* El rombo negro implica idénticos tiempos de vida, y el rombo blanco implica que se trata de una referencia o puntero a objeto (tiempo de vida no idéntico)

47

## Reto 7: ampliación o reducción del servidor web

- Objetivo: permitir cambios futuros en sistemas con restricciones de recursos o para cumplir progresivamente objetivos del desarrollo
- Solución: familias de programas y subconjuntos (se aplica sobre todo en infraestructura *middleware* reutilizable)
- Familias de programas:
  - Se basan en identificar subconjuntos mínimos del sistema que son útiles
  - También se identifican los incrementos mínimos que se pueden aplicar a los subconjuntos para aumentar su utilidad

48

## Ejemplos de familias de programas

- Servicios diferentes para mercados diferentes (alfabetos, formatos E/S)
- Plataformas hardware o software diferentes (compiladores, sistemas operativos)
- Distintas prioridades (velocidad frente a área/volumen)
- Recursos internos diferentes (librerías, estructuras de datos)
- Eventos externos diferentes (interfaz UNIX dispositivos E/S)
- Compatibilidad hacia atrás (mantener errores?!)

- Ejemplo: JAWS (web server) y TAO (implementación del *middleware* CORBA) se desarrollaron usando el toolkit ACE

49

## Contenidos

- Métodos de diseño software. Criterios de evaluación
- Diseño funcional frente a diseño orientado a objetos
- Aspectos clave en el diseño software. Ejemplos
- **Reglas de diseño**
- El lenguaje C++. Objetivos y características principales

50

## Reglas de diseño (1/4)

- Asegurarse de que el problema está bien definido
  - Verificar que criterios, requisitos y restricciones están definidos
- El qué va antes que el cómo
  - P.e. definir los servicios antes que las estructuras que los realizarán
- Separar ámbitos ortogonales (no conectar lo independiente)
- Diseñar la funcionalidad externa antes que la interna
- Separar problemas complejos en sencillos que se resuelven independientemente

51

## Reglas de diseño (2/4)

- Trabajar a varios niveles de abstracción simultáneamente
- Diseñar facilitando las ampliaciones
  - P.e. resolver problemas generales y no un caso concreto, no incluir lo que es intangible, no restringir lo irrelevante
- Usar prototipado rápido de alto nivel antes de implementar
- Los detalles dependen de las abstracciones y no al revés
- La granularidad de desarrollo y de reutilización es la misma
  - Sólo componentes con un desarrollo monitorizado/documentado sistemáticamente pueden ser reutilizados efectivamente

52

## Reglas de diseño (3/4)

- Las clases dentro de un componente deben constituir un grupo cerrado (los cambios futuros serán internos al grupo) y deben reutilizarse juntas
- La estructura de dependencias en un componente es un DAG y debe ir en la dirección de aumentar la estabilidad
- Un componente completamente estable sólo debe incluir clases abstractas
- Si es posible, resolver los problemas con patrones de diseño

53

## Reglas de diseño (4/4)

- Si se cruzan paradigmas, definir una interfaz que los separe
- Las entidades software deben seguir el principio abierto/cerrado (para ampliación/modificación)
- Las clases derivadas deben poder usarse con la interfaz de la clase base sin que el usuario conozca la diferencia
- Primero hacer que funcione, luego hacer que funcione rápido
- Mantener la consistencia entre representaciones (deben ser equivalentes a medida que la implementación se optimiza)

54

# Errores de diseño habituales

- Empezar considerando sólo parte de los requisitos
- Refinar directamente la especificación de requisitos
- No considerar cambios potenciales
- Hacer un diseño demasiado detallado (sobrerrestringido)
- Usar especificaciones ambiguas (resultado insatisfactorio)
- No documentar las decisiones de diseño
- Diseño inconsistente (desarrollo por separado no integrable)

55

# Contenidos

- Métodos de diseño software. Criterios de evaluación
- Diseño funcional frente a diseño orientado a objetos
- Aspectos clave en el diseño software. Ejemplos
- Reglas de diseño
- **El lenguaje C++. Objetivos y características principales**

56

# Panorámica del lenguaje C++

- C++ (Bjarne Stroustrup, AT&T Bell Labs, años 80) es una extensión compatible con C que permite:
  - Diseño orientado a objetos (clases, herencia, métodos virtuales)
  - Abstracción de datos (encapsulado, mecanismo de clases)
  - Programación genérica (tipos parametrizados)
  - Manejo de errores (excepciones)
  - Identificación de tipo de objeto en tiempo de ejecución (RTTI)
- C++ no permite describir explícitamente
  - Concurrencia (procesos que se ejecutan simultáneamente)
  - Persistencia (objetos no tienen una representación en disco)
  - *Garbage collection* (liberación de espacio de disco)
  - Distribución (ejecución en varias máquinas)

57

# Ejemplo: pila en C (1/2)

- Implementación "cruda" de una pila (*stack*) en C:

```
typedef int T;
#define MAX_STACK 100 /* const int MAX_STACK = 100; */
T stack[MAX_STACK];
int top = 0;
T item = 10;
stack[top++] = item; // push
...
item = stack[--top]; // pop
```

- Interfaz definida en **`stack.h`** e implementada en **`stack.c`**:

```
/* Type of Stack element. */
typedef int T;

/* Stack interface. */
int create (int size);
int destroy (void);
void push (T new_item);
void pop (T *old_top);
void top (T *cur_top);
int is_empty (void);
int is_full (void);
```

```
#include "stack.h"
static int top_, size_; /* Hidden within this file. */
static T *stack_;
int create (int size) {
  top_ = 0; size_ = size;
  stack_ = malloc (size * sizeof (T));
  return stack_ == 0 ? -1 : 0;
}
void destroy (void) { free ((void *) stack_); }
void push (T item) { stack_[top_++] = item;}
void pop (T *item) { *item = stack_[--top_]; }
void top (T *item) { *item = stack_[top_ - 1]; }
int is_empty (void) { return top_ == 0; }
int is_full (void) { return top_ == size_; }
```

# Ejemplo: pila en C (2/2)

- Ejemplo de utilización:

```
#include "stack.h"
void foo (void) {
  T i;
  push (10); /* Oops, forgot to call create! */
  push (20);
  pop (&i);
  destroy ();
}
```

- Problemas:
  - Hay que llamar `create()` para empezar y `destroy()` al acabar
  - Sólo hay una pila y un único tipo de pila
  - Contaminación del espacio de nombres
  - Implementación no reentrante



# Ejemplo: pila como dato abstracto en C (1/2)

- stack.h:

```
typedef int T;
typedef struct { size_t top_, size_; T *stack_; } Stack;

int Stack_create (Stack *s, size_t size);
void Stack_destroy (Stack *s);
void Stack_push (Stack *s, T item);
void Stack_pop (Stack *, T *item);
/* Must call before pop'ing */
int Stack_is_empty (Stack *);
/* Must call before push'ing */
int Stack_is_full (Stack *);
/* ... */
```

- stack.c:

```
#include "stack.h"
int Stack_create (Stack *s, size_t size) {
  s->top_ = 0; s->size_ = size;
  s->stack_ = malloc (size * sizeof (T));
  return s->stack_ == 0 ? -1 : 0;
}
void Stack_destroy (Stack *s) {
  free ((void *) s->stack_);
  s->top_ = 0; s->size_ = 0; s->stack_ = 0;
}
void Stack_push (Stack *s, T item)
{ s->stack_[s->top_++] = item; }
void Stack_pop (Stack *s, T *item)
{ *item = s->stack_[--s->top__]; }
int Stack_is_empty (Stack *s) { return s->top_ == 0; }
```

# Ejemplo: pila como dato abstracto en C (2/2)

- Utilización:

```
void foo (void) {
  Stack s1, s2, s3; /* Multiple stacks! */
  T item;
  Stack_pop (&s2, &item); /* Pop'd empty stack */
  /* Forgot to call Stack_create! */
  Stack_push (&s3, 10);
  s2 = s3; /* Disaster due to aliasing!!! */
  /* Destroy uninitialized stacks! */
  Stack_destroy (&s1); Stack_destroy (&s2);
}
```

- Problemas:
  - No garantiza la inicialización, la terminación, ni la asignación
  - Sólo hay un único tipo de pila
  - Mucho *overhead* debido a las llamadas a función
  - No hay un tratamiento de errores generalizado
  - El compilador no fuerza la ocultación de información:

```
s1.top_ = s2.stack_[0]; /* Violate abstraction */
s2.size_ = s3.top_; /* Violate abstraction */
```

61

# Ejemplo: pila como dato abstracto en C++ (1/2)

- Definición:

```
typedef int T;
class Stack {
public:
  Stack (size_t size);
  Stack (const Stack &s);
  void operator= (const Stack &);
  ~Stack (void);
  void push (const T &item);
  void pop (T &item);
  int is_empty (void) const;
  int is_full (void) const;
private:
  size_t top_, size_;
  T *stack_;
};
```

- Implementación:

```
Stack::Stack (size_t s): top_ (0), size_ (s), stack_ (new T[s]) {}
Stack::Stack (const Stack &s):
  : top_ (s.top_), size_ (s.size_), stack_ (new T[s.size_]) {
  for (size_t i = 0; i < s.size_; i++) stack_[i] = s.stack_[i];
}
void Stack::operator = (const Stack &s) {
  if (this == &s) return;
  delete [] stack_;
  stack_ = new T[s.size_]; top_ = s.top_; size_ = s.size_;
  for (size_t i = 0; i < s.size_; i++) stack_[i] = s.stack_[i];
}
Stack::~Stack (void) { delete [] stack_; }
int Stack::is_empty (void) const { return top_ == 0; }
int Stack::is_full (void) const { return top_ == size_; }
void Stack::push (const T &item) { stack_[top_++] = item; }
void Stack::pop (T &item) { item = stack_[--top_]; }
```

## Ejemplo: pila como dato abstracto en C++ (2/2)

- Utilización:

```
#include "Stack.h"
void foo (void) {
  Stack s1 (1), s2 (100);
  T item;
  if (!s1.is_full ())
    s1.push (473);
  if (!s2.is_full ())
    s2.push (2112);
  if (!s2.is_empty ())
    s2.pop (item);
  // Access violation caught at compile-time!
  s2.top_ = 10;
  // Termination is handled automatically.
}
```

- Ventajas:
  - Ocultación y abstracción:

```
Stack s1 (200);
s1.top_ = 10 // Error flagged by compiler!
```

  - Inicialización y terminación:

```
Stack s1 (1000); // constructor called automatically
// ...
// Destructor called automatically
```

- Inconvenientes: manejo de excepciones, único tipo de pila, llamadas a funciones (usar `inline` reduce el *overhead*)

63

## Ejemplo: pila con excepciones en C++ (1/2)

- Stack.h:

```
typedef int T;
class Stack {
public:
  class Underflow { /* ... */ };
  class Overflow { /* ... */ };
  Stack (size_t size);
  ~Stack (void);
  void push (const T &item) throw (Overflow);
  void pop (T &item) throw (Underflow);
private:
  size_t top_, size_;
  T *stack_;
};
```

- Stack.cpp:

```
void Stack::push (const T &item) throw (Stack::Overflow)
{
  if (is_full ())
    throw Stack::Overflow ();
  stack_[top_++] = item;
}

void Stack::pop (T &item) throw (Stack::Underflow)
{
  if (is_empty ())
    throw Stack::Underflow ();
  item = stack_[--top_];
}
```

64

# Ejemplo: pila con excepciones en C++ (2/2)

- Utilización:

```
#include "Stack.h"
void foo (void) {
  Stack s1 (1), s2 (100);
  try {
    T item;
    s1.push (473);
    s1.push (42); // Exception, push'd full stack!
    s2.pop (item); // Exception, pop'd empty stack!
    s2.top_ = 10; // Access violation caught!
  } catch (Stack::Underflow) { /* Handle underflow... */ }
  catch (Stack::Overflow) { /* Handle overflow... */ }
  catch (...) { /* Catch anything else... */ throw; }
  }
  // Termination is handled automatically.
}
```



# Ejemplo: pila mediante *template* en C++ (1/2)

- Definición:

```
template <typename T> class Stack {
public:
  Stack (size_t size);
  ~Stack (void)
  void push (const T &item);
  void pop (T &item);
  int is_empty (void);
  int is_full (void);
private:
  size_t top_, size_;
  T *stack_;
};
```

- Implementación: (sin excepciones)

```
template <typename T> inline
Stack<T>::Stack (size_t size)
  : top_ (0), size_ (size), stack_ (new T[size]) { }

template <typename T> inline
Stack<T>::~Stack (void) { delete [] stack_; }

template <typename T> inline void
Stack<T>::push (const T &item) { stack_[top_++] = item; }

template <typename T> inline void
Stack<T>::pop (T &item) { item = stack_[--top_]; }
```

## Ejemplo: pila mediante *template* en C++ (2/2)

- Utilización:

```
#include "Stack.h"

void foo (void) {
  Stack<int> s1 (1000);
  Stack<float> s2;
  Stack< Stack <Activation_Record> *> s3;

  s1.push (-291);
  s2.top_ = 3.1416; // Access violation caught!
  s3.push (new Stack<Activation_Record>);
  Stack <Activation_Record> *sar;
  s3.pop (sar);
  delete sar;
  // Termination is handled automatically
}
```

- Problemas y soluciones:
  - Recompilar por cambios en implementación =>desacoplar de la interfaz con herencia + *dynamic binding* (asociar llamada a método)
  - Las extensiones implican acceder al código fuente => usar clases base abstractas

67

## Ejemplo: implementación orientada a objeto de pila en C++ (1/4)

- Clase base abstracta: métodos virtuales puros

```
template <typename T>
class Stack {
public:
  virtual void push (const T &item) = 0;
  virtual void pop (T &item) = 0;
  virtual int is_empty (void) const = 0;
  virtual int is_full (void) const = 0;
  void top (T &item) { // Template Method
    pop (item); push (item);
  }
};
```

- Los métodos virtuales puros garantizan que el compilador no permitirá la instanciación de la clase base abstracta

68

## Ejemplo: implementación orientada a objeto de pila en C++ (2/4)

- Especialización:
  (*bounded stack*)

```
#include "Stack.h"
#include "vector"

template <typename T> class B_Stack : public Stack<T> {
public:
  enum { DEFAULT_SIZE = 100 };
  B_Stack (size_t size = DEFAULT_SIZE);
  virtual void push (const T &item);
  virtual void pop (T &item);
  virtual int is_empty (void) const;
  virtual int is_full (void) const;
private:
  size_t top_; // built-in
  std::vector<T> stack_; // user-defined
};
```

```
template <typename T>
B_Stack<T>::B_Stack (size_t size): top_ (0), stack_ (size) {}

template <typename T> void
B_Stack<T>::push (const T &item) { stack_[top_++] = item; }

template <typename T> void
B_Stack<T>::pop (T &item) { item = stack_[--top_]; }

template <typename T> int
B_Stack<T>::is_full (void) const { return top_ >= stack_.size (
```

## Ejemplo: implementación orientada a objeto de pila en C++ (3/4)

- Especialización:
  (*unbounded stack*)

```
template <typename T> class Node; // forward declaration.
template <typename T> class UB_Stack : public Stack<T> {
public:
  enum { DEFAULT_SIZE = 100 };
  UB_Stack (size_t hint = DEFAULT_SIZE);
  ~UB_Stack (void);
  virtual void push (const T &new_item);
  virtual void pop (T &top_item);
  virtual int is_empty (void) const { return head_ == 0; }
  virtual int is_full (void) const { return 0; }
private:
  // Head of linked list of Node<T>'s.
  Node<T> *head_;
};
```

```
template <typename T> UB_Stack<T>::UB_Stack (size_t): head_ (0)

template <typename T> void UB_Stack<T>::push (const T &item) {
  Node<T> *t = new Node<T> (item, head_); head_ = t;
}

template <typename T> void UB_Stack<T>::pop (T &top_item) {
  top_item = head_->item_;
  Node<T> *t = head_; head_ = head_->next_;
  delete t;
}

template <typename T> UB_Stack<T>::~UB_Stack (void)
{ for (T t; head_ != 0; pop (t)) continue; }
```

```
template <typename T> class Node {
friend template <typename T>
  class UB_Stack;
public:
  Node (T i, Node<T< *n = 0):
    item_ (i), next_ (n) {}
private:
  T item_;
  Node<T> *next_;
}
```

clase Node

# Ejemplo: implementación orientada a objeto de pila en C++ (4/4)

- Utilización:
  - Código que no depende de la implementación

```
template <typename T> Stack<T> *make_stack (int use_B_Stack) {
  if (use_B_Stack) return new B_Stack<T>;
  else return new UB_Stack<T>;
}

void foo (Stack<int> *stack) {
  int i;
  stack->push (100);
  stack->pop (i);
  // ...
}

foo (make_stack<int> (0));
```

  - También es posible hacer cambios durante la ejecución sin tener que recompilar (p.e. selección del tipo de pila)

71

# Resumen

- El diseño orientado a objetos presenta múltiples ventajas (modularidad, reutilización, ampliación) frente al diseño funcional (intuitivo, apropiado para programas pequeños)

- En el diseño de componentes software es fundamental separar la interfaz (lo común) de la implementación (lo variable)

- Las principales contribuciones de C++ son que permite tanto la definición de tipos de datos abstractos como la programación genérica (p.e. usar iteradores frente a suponer una estructura de datos concreta)

72