# Parte I: El computador y el proceso de programación

- 1.Introducción a los computadores y su programación
- 2. Introducción al análisis y diseño de algoritmos
- 3. Introducción al análisis y diseño de programas
- 4. Verificación de programas



## Notas:

**1. Introducción a los computadores y su programación**

- Arquitectura básica de un computador.El software del sistema. Lenguajes de Alto Nivel. El proceso de compilación. El ciclo de vida del software.

**2. Introducción al análisis y diseño de algoritmos.**

- Diseño de un programa. Concepto de algoritmo. Descripción de algoritmos: el pseudolenguaje. Tiempo de ejecución de algoritmos. La notación O(n). Ejemplos de análisis.

**3. Introducción al análisis y diseño de programas**

- Actividades del ciclo de vida del software. Paradigmas de desarrollo de programas. Análisis y especificación. Diseño arquitectónico. Técnicas de diseño detallado.

**4. Verificación de programas**

- Importancia de la verificación. Estrategias de prueba. Depuración. Elección de datos para la prueba.

# 1. Actividades del ciclo de vida del software

## Análisis y Especificación

## Diseño de la Arquitectura

## Diseño Detallado

## Codificación y Desarrollo

## Integración y Verificación o Prueba

## Operación y Mantenimiento



## Notas:

*Análisis y especificación*. En esta etapa se analiza la naturaleza del problema, y se establecen los requerimientos del sistema. En la especificación se desarrollan las descripciones del sistema, de sus restricciones, y de los recursos necesarios.

*Diseño de la Arquitectura*. A partir de las especificaciones funcionales, se diseña la estructura del sistema que resuelve el problema. Esta estructura representa las partes importantes del sistema y sus relaciones, así como las estructuras de datos más importantes.

*Diseño Detallado.* Las partes importantes del sistema se diseñan en detalle, describiendo los algoritmos y estructuras de datos concretas. Se detallan las interfaces entre las diferentes partes. Las etapas de diseño suelen requerir varios niveles de refinamiento.

*Codificación y Desarrollo*. Producción de una realización física del diseño.

*Integración y Verificación*. Suelen ser fases cíclicas, en las que de forma incremental se van probando e integrando las diversas partes del sistema.

*Operación y Mantenimiento*. Reparación de problemas, adaptación a nuevas condiciones, mejoras, etc.

# Actividades en Cada Etapa

| Etapa | Entradas | Salidas |
|---|---|---|
| Análisis | Necesidades iniciales, contexto, problemas del usuario | Definición de requerimientos |
| Especificación | Requerimientos, contexto del sistema | Especificaciones funcionales, diseño externo del sistema |
| Diseño de la Arquitectura | Especificaciones, contexto, experiencia previa | Definición de módulos e interfaces |
| Diseño Detallado | Descripción de la arquitectura, detalles del entorno | Descripción de la estructura de los programas |



## Notas:

Análisis:

- Identificación de las funciones más importantes, recolección de información y de restricciones.

Especificación:

- Conversión de las necesidades en funciones explícitas, selección de las restricciones que son operativas, descripción de las interfaces al usuario.

Diseño de la Arquitectura:

- Determinar la estructura del problema, identificar las partes importantes del sistema, establecer las relaciones entre las partes, abstracción, y descomposición.

Diseño Detallado:

- Abstracción, elaboración, selección de alternativas.

# Actividades en Cada Etapa (cont.)

| Etapa | Entradas | Salidas |
|---|---|---|
| Codificación y Desarrollo | Descripción de la arquitectura, detalles del entorno | Código de unidades de programa y documentación |
| Integración y Verificación | Descripción de la arquitectura y código de las unidades de programa con su documentación | Código del programa o programas completos, ficheros de datos, sistema completo |
| Operación y Mantenimiento | Documentación del sistema, requerimientos de operación, peticiones de cambios | Sistema mejorado |



## Notas:

Codificación y Desarrollo:

- Codificación de algoritmos y estructuras de datos

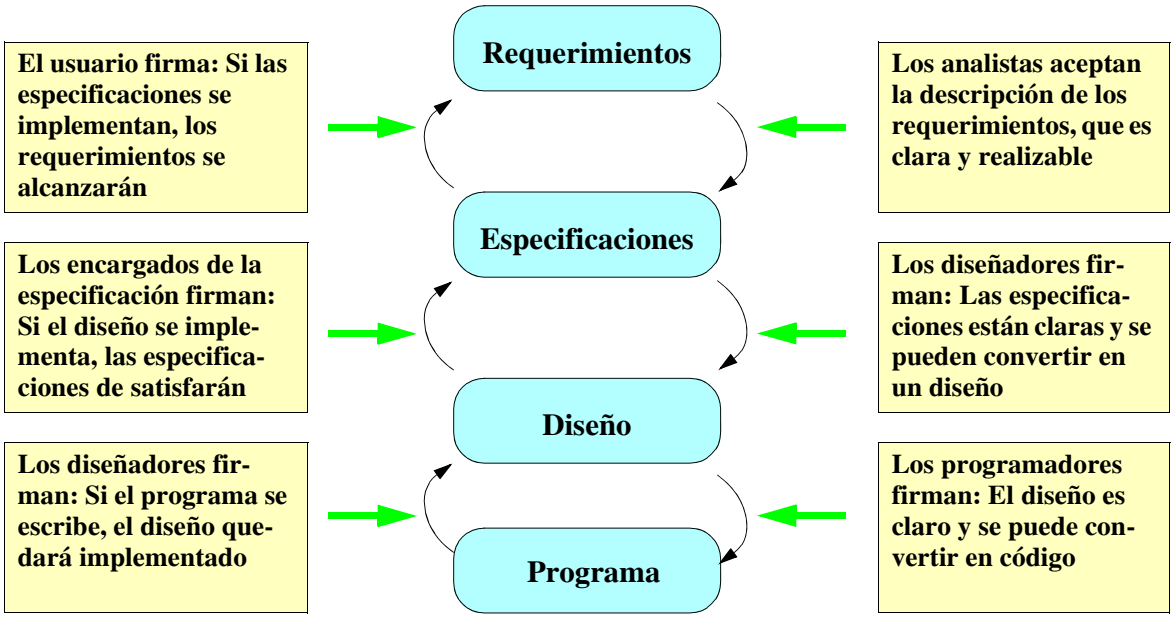
Integración y Verificación:

- Depuración y verificación de unidades de programa y de prototipos del sistema global cada vez más elaborados, hasta llegar al sistema completo

Operación y mantenimiento:

- Reprogramación, mejora, depuración del rediseño, etc.

# Validación Paso a Paso del Proceso






## Notas:

Cuando se trabaja en asociación con otras personas para realizar un proyecto relativamente grande y utilizando técnicas avanzadas, debe de existir una filosofía y metodología de dirección que guíe el proyecto, para su terminación con éxito. Este éxito será fuertemente dependiente de la calidad del proceso de dirección.

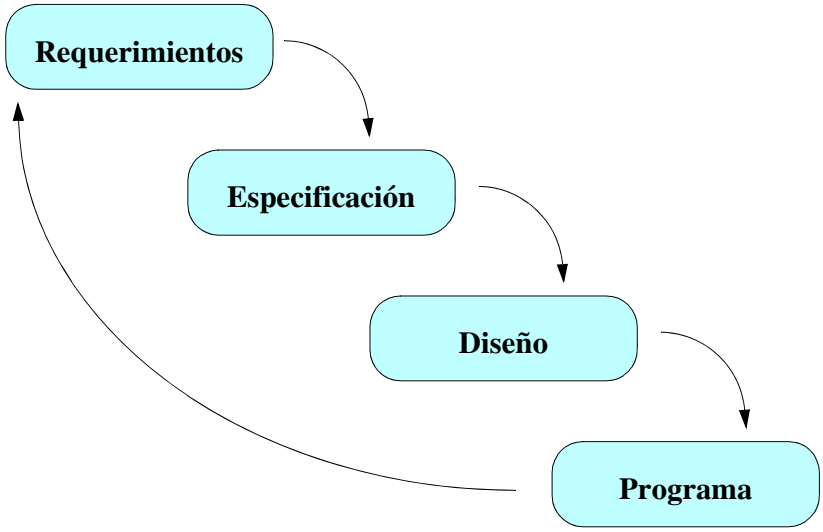
El software desarrollado puede ser interno, para uso por el propio organismo que lo desarrolla, o externo, realizado para ser utilizado por otras personas. En ambos casos, las técnicas de dirección empleadas deben de ser muy similares (aunque en muchas ocasiones en el desarrollo interno esta medida no se tenga en cuenta).

Es importante, en ambos casos, tener presente todo el ciclo de vida, teniendo especial cuidado con los límites entre las distintas fases.

Cada uno de los límites entre las fases de desarrollo deberá estar claramente representado mediante un documento que deberá establecer claramente lo que se ha realizado en la etapa precedente. Esto, junto con una revisión del diseño, ayudará a la evaluación y validación.

En la figura se muestra el método de validación paso a paso del proceso de desarrollo. Al comienzo de cada etapa se documenta y se cierra la etapa anterior, de modo que se minimiza la cantidad de iteración y de cambios estructurales.

# Validación por Comparación con los Requerimientos

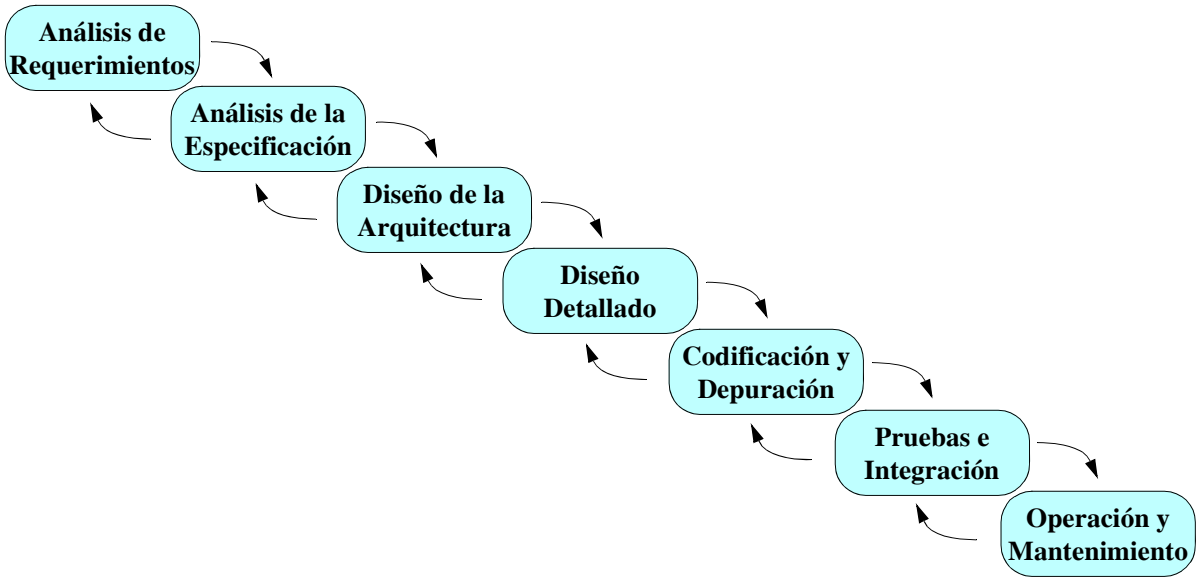





## Notas:

En la figura se muestra el método de validación del programa por comparación con los requerimientos. Este caso, en el que se utiliza el programa para realizar la validación, llevará desafortunadamente a que cualquier error serio detectado en la validación implique un costo muy elevado, al estar el programa ya muy avanzado.

## Paradigma clásico (en cascada) del ciclo de vida del software:






## Notas:

El paradigma clásico constituye la herramienta actual más potente para el desarrollo del software. Se basa en la secuencia de pasos lógicos que aparece en la figura, y que son optimizados a nivel individual, uno a uno.

Actualmente se considera que este paradigma limita la calidad del proceso, por las siguientes razones:

1. Exige la toma de decisiones en fases iniciales, sin conocer su efecto en fases sucesivas
2. Considera que los requerimientos del programa están definidos antes de que éste haya sido diseñado.
3. Considera la especificación del programa como algo previo y no interaccionante con el diseño.
4. Se basa en criterios tales como la experiencia del programador, y es difícil de automatizar
5. El paradigma hace muy difícil y costoso el proceso de mantenimiento del software.

Sin embargo, el uso de un paradigma como este reporta muchos beneficios con respecto a un proceso desordenado, en el que no se siga una metodología clara.

# Objetivos de los Nuevos Paradigmas

## Los nuevos paradigmas tratan de:

- Proporcionar a los usuarios un objeto ejecutable en las etapas iniciales del proceso
- Analizar y reducir el riesgo
- Automatizar la producción del software

## Se pueden utilizar en combinación con el paradigma clásico, o una combinación de varios de ellos



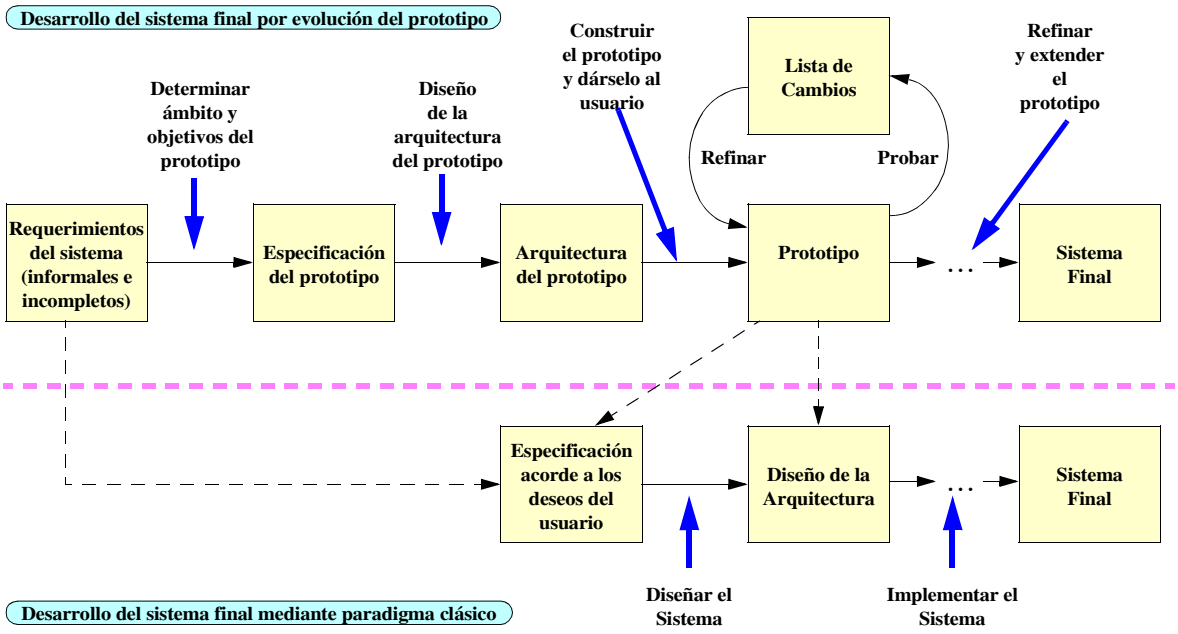
## Notas:

Objetivos de los nuevos paradigmas:

1. Automatizar el proceso de desarrollo del programa
2. Minimizar los errores cometidos en las etapas iniciales del desarrollo
3. Posibilitar que el usuario actúe como analista
4. Analizar y reducir el riesgo
5. Eliminar las tareas rutinarias en las que se posibilite introducir errores
6. Incrementar la capacidad de optimización
7. Facilitar el mantenimiento
8. Mantener actualizada la especificación

En la práctica, es habitual mezclar varios paradigmas, uno para cada una de las fases del proyecto, con objeto de obtener las mejores ventajas de cada uno. Por ejemplo, es habitual hacer las fases iniciales del proyecto según la metodología orientada a prototipo, y luego seguir con una metodología clásica para llegar al programa final.

# Paradigma Basado en Prototipo






## Notas:

En el paradigma basado en prototipo se construye a partir de las especificaciones un modelo de trabajo del programa que se está desarrollando.

El objetivo del prototipo es clarificar las características y operaciones de un sistema que se está desarrollando, practicando con un modelo previamente construido.

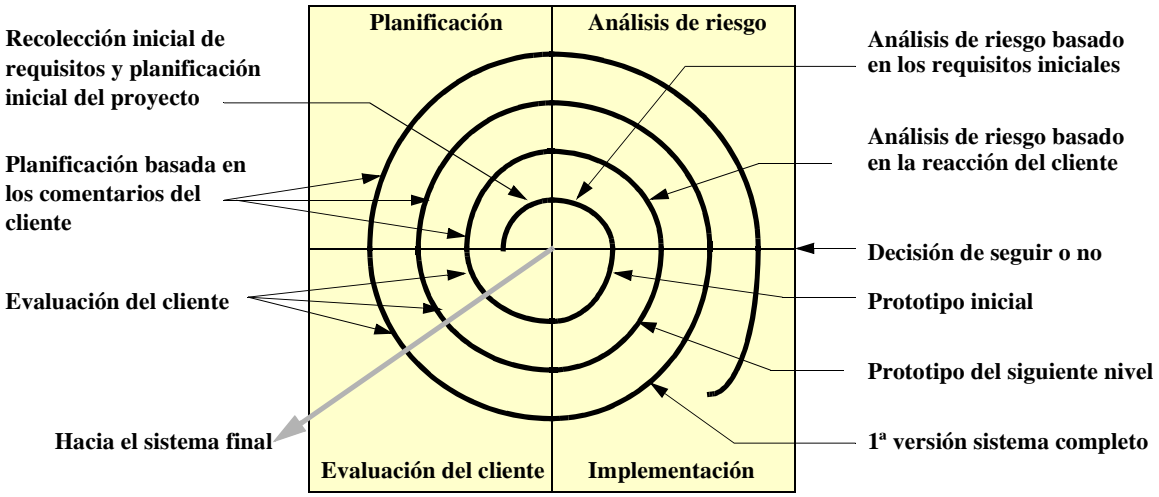
Los prototipos pueden generase reduciendo el tamaño o la funcionalidad.

Los prototipos ayudan a establecer los requerimientos del usuario cuando el sistema no es aún bien comprendido. (“Yo sabré lo que quiero cuando lo vea”).

Cuando el sistema a nivel de prototipo ha sido aceptado, es necesario definir cómo continuar:

- Transformar el prototipo en un sistema completamente operativo. Se corre el peligro de que el diseño del prototipo no sea adecuado al producto final, por lo que éste no tendrá la calidad suficiente
- Iniciar un proceso de desarrollo clásico basándose en la experiencia del prototipo. Es una aproximación más costosa, pero normalmente más segura.

# Paradigma de desarrollo en espiral






## Notas:

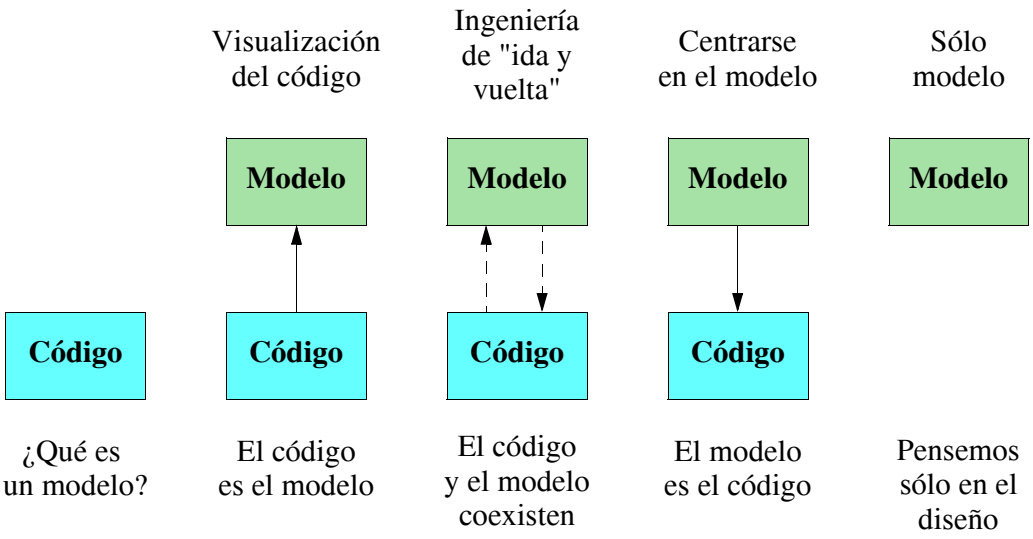
El modelo en espiral para la ingeniería de software fue ideado por B. Bohem (1988) y trata de agrupar las mejores ideas del paradigma clásico y del basado en prototipos, añadiendo además un nuevo elemento que faltaba en ambos: el análisis de riesgos.

El modelo en espiral trata de realizar un desarrollo evolutivo del producto, desde un prototipo inicial hasta llegar al sistema final, a través de varias etapas. En cada una de las etapas se siguen, por este orden, las siguientes cuatro actividades principales:

- *Planificación*: Determinación de objetivos, alternativas, y restricciones, de acuerdo con la etapa actual. Salvo en la primera etapa, se tendrán en cuenta las valoraciones del cliente.
- *Análisis de riesgo*: Análisis de las alternativas, e identificación y valoración de riesgos. Como resultado se eligen las alternativas de menor riesgo, o si el riesgo es demasiado alto se abandona el proyecto.
- *Implementación*: Desarrollo del producto del siguiente nivel
- *Evaluación del cliente*: valoración de los resultados por parte del cliente

En la representación gráfica del modelo en espiral, la curva se sitúa más cercana al centro en las primeras etapas del diseño, y se aproxima a los bordes en las etapas finales. El borde representa el sistema final.

# Diseño basado en modelos






## Notas:

El modelado tiene una tradición muy importante en la ingeniería del software

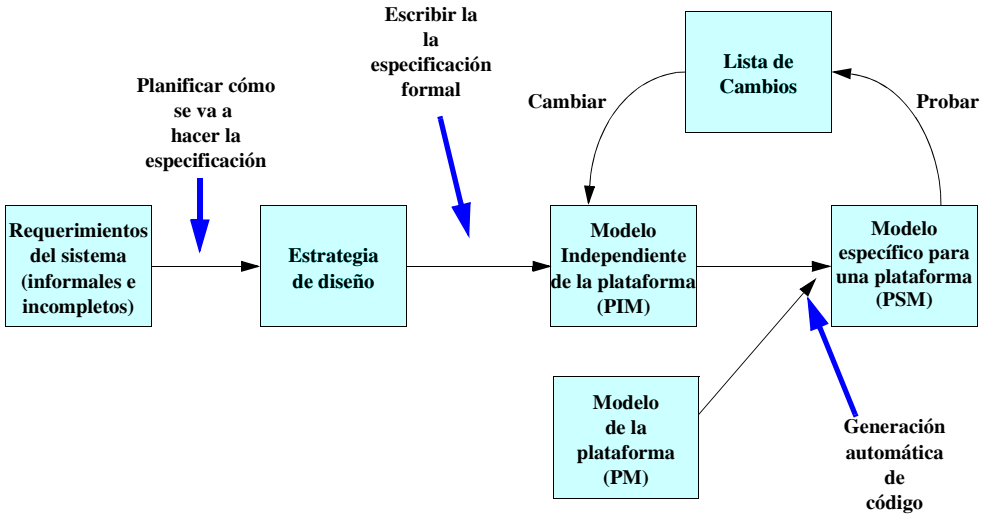
Antiguamente se usaba el modelado sólo para tener una idea de cómo era el diseño del código

Actualmente se utiliza UML para hacer el diseño, y luego se hace el código de acuerdo a ese diseño. A veces es difícil mantenerlos coherentes porque la transformación es manual

A medida que se dispone de herramientas de transformación de modelos es posible centrarse en el modelo y generar casi todo el código de manera automática. Sólo algunas partes pequeñas necesitarán codificarse a mano

En el futuro, probablemente se trabajará sólo a nivel de modelo.

# Técnicas de diseño dirigido por modelos (MDD, MDA, MDE)






## Notas:

Las técnicas de diseño basado en modelos en el uso de lenguajes de modelado (principalmente UML, lenguaje universal de modelado) que permiten realizar una especificación funcional del programa a alto nivel, de manera independiente de la plataforma (modelo PIM). Usando luego una herramienta de generación de código que parte del modelo de la plataforma (PM) y de la especificación funcional (PIM) se generar el programa automáticamente para una plataforma concreta (PSM). Posteriormente el programa puede ser evaluado, y los cambios que se realicen en la especificación se verán implementados de forma automática. Ello permite además adaptarse fácilmente a varias plataformas diferentes.

El desarrollo del modelo independiente del lenguaje es la principal dificultad para la generalización de este método. La especificación formal debe presentar los siguientes aspectos contradictorios:

- *formal*, para que admita con éxito las transformaciones automáticas.
- *comprensible* para el usuario que debe validar el comportamiento previsto del sistema.

La especificación formal es el aspecto central de este paradigma:

- La especificación formal es el único aspecto que es validado respecto de las especificaciones del usuario
- Es el punto de arranque de la generación automática de código
- Cuando se mantiene el programa, este mantenimiento se lleva a cabo a partir de la modificación de la especificación formal.

# 2. Introducción a la Etapa de Análisis y Especificación

## Objetivos:

- **Formular un conjunto correcto y completo de los requerimientos del sistema**
- **Producir un conjunto de especificaciones, que:**
  - **describe el comportamiento externo**
  - **describe restricciones sobre la implementación**
  - **debe servir de referencia para los usuarios**
  - **debe servir de referencia para el mantenimiento**
  - **describe las respuestas aceptables ante situaciones anómalas**



## Notas:

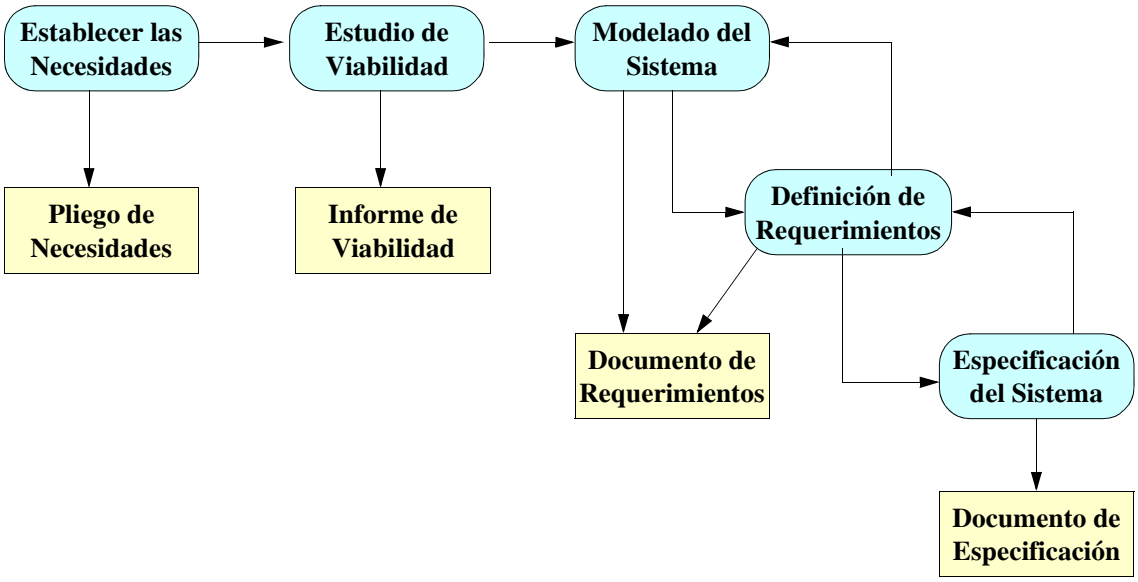
Uno de los problemas clave de cualquier diseño software o hardware es el formular un conjunto correcto y completo de los requerimientos del sistema. A partir de estos requerimientos deberá producirse un conjunto de especificaciones que, cuando sea propiamente implementado, verificará los requerimientos.

A menudo es difícil determinar donde acaban los requerimientos y donde comienzan las especificaciones. La idea es que los requerimientos indican “qué se ha de conseguir”, mientras que las especificaciones muestran “cómo se va a conseguir”.

Inicialmente se realiza una formalización (en un documento) de las necesidades.

A partir de estas necesidades se elaboran los documentos de requerimientos, especificaciones, diseño inicial, etc.

# Modelo de Formulación de Requerimientos






## Notas:

Existen dos destinatarios del proceso de análisis de requerimientos y especificación. Unos son los potenciales usuarios del sistema. Estos exigen una definición de requerimientos que indique las funciones del sistema sin detalles de implementación. Los otros son los diseñadores del sistema, que requieren una especificación con abundancia de detalles y restricciones.

El proceso de análisis comienza con un documento que describe brevemente las necesidades, y un estudio de viabilidad. Posteriormente se modela el sistema a programar y se describen todos sus requerimientos detallados, generándose los siguientes documentos:

- *Documento de Requerimientos*. Es una definición en lenguaje natural de los servicios que el usuario del sistema va a recibir. Debe estar escrito en lenguaje no técnico, de forma que los gestores del desarrollo y los clientes puedan comprender su alcance. Este documento, debe ser lo suficientemente preciso para que sobre él se establezca la relación contractual.
- *Especificación del diseño*. Es una descripción formal y detallada que sirve de base para el diseño e implementación del software. Va a ser utilizado por los diseñadores, pero debe de ser también comprensible para el usuario.

En general, los requerimientos expresan lo que el sistema debe hacer, sin explicar cómo, y la especificación indica cómo se va a hacer, aunque sin entrar en detalles de la implementación informática (que forman parte de la siguiente etapa, la de diseño).

# Estructura del Documento de Requerimientos

1. Introducción
2. Características del computador y del sistema
3. Descripción de la información
4. Descripción de las funciones del software
5. Descripción del comportamiento ante sucesos diversos
6. Criterios de validación
7. Cambios



## Notas:

1. Introducción. Modelo conceptual del sistema. Principios de organización, resúmenes de otras secciones, guía de notación, etc.
2. Características del computador y del sistema. Si el computador está predeterminado, una descripción general. En caso contrario un sumario de las características requeridas. Interfaces Hardware. Concisa descripción de la información recibida o transmitida por el computador.
3. Descripción de la Información: representación del flujo de información. Contenido de la información y su representación.
4. Descripción de las funciones del software. Qué debe de hacer el software en diversas situaciones y respuesta ante eventos. Restricciones y requisitos de rendimiento.
5. Descripción del comportamiento ante sucesos diversos. Respuesta a eventos indeseados. Qué debe de hacer el software ante errores, fallos de alimentación, etc.
6. Criterios de validación. Límites de rendimiento. Clases de pruebas a realizar. Consideraciones especiales.
7. Cambios. Tipos de cambios realizados o por realizar.

# Análisis Orientado a Objetos

UNIVERSIDAD DE CANTABRIA

**Las metodologías orientadas a objetos son cada vez más populares, porque**

- **facilitan la reutilización de código (y de su diseño)**
- **son fáciles de comprender, modificar, y extender**

**En el análisis orientado a objetos se realiza la especificación del sistema descomponiéndolo en objetos o clases de objetos**

**Cada objeto o clase tiene tres componentes:**

- **nombre: identifica al objeto**
- **atributos: valores que almacena**
- **operaciones: modifican los atributos y pueden invocar operaciones de otros objetos**



## Notas:

UNIVERSIDAD DE CANTABRIA

Desde principio de los años 80 se han venido generalizando los métodos orientados a objeto tanto para el análisis y diseño, como para la codificación de programas. El método hace el software más fácil de entender, al existir una relación directa entre su estructura y la del sistema o problema a resolver. Asimismo, el encapsulamiento de todos los aspectos relacionados con cada objeto en un mismo lugar hacer el software más fácil de modificar, de extender, y de ser reutilizado.

En un sistema orientado a objetos los objetos que tienen las mismas características se agrupan en las llamadas clases de objetos. Por ejemplo, en un proceso industrial, un sensor de posición puede ser una clase de objetos, ya que puede haber muchos sensores de posición en diversos lugares de la maquinaria. Cada uno de los sensores individuales será un objeto.

En una metodología orientada al objeto las clases de objetos o los objetos individuales suelen tener tres componentes:

- *nombre*: identifica al objeto o a la clase
- *atributos*: existe un atributo por cada uno de los datos o valores que el objeto puede almacenar; el conjunto de los valores de los atributos de un objeto determina su estado
- *operaciones*: cada objeto tiene un conjunto de operaciones que se pueden invocar desde el exterior; al invocarse una operación se le pueden pasar parámetros (datos de entrada o salida) al objeto; la ejecución de la operación normalmente implica el cambio de los atributos y, posiblemente, la invocación de operaciones de otros objetos.

# Extensión de Objetos y Herencia

**Uno de los conceptos fundamentales en las técnicas orientadas al objeto es la posibilidad de extender un objeto o una clase para obtener otro similar.**

- **el objeto o clase extendido hereda todos los atributos y operaciones de su antecesor**
- **además, puede añadir más atributos y operaciones**

**Otro de los conceptos fundamentales es la comunicación entre objetos, que se realiza al invocar sus operaciones**

**Por tanto, una metodología orientada a objetos se basa en:**

- **objetos + clases + herencia + comunicación**



## Notas:

Los objetos o las clases se pueden derivar unos de otros, si presentan elementos comunes. Al crear un objeto o clase como extensión de otro, el nuevo objeto suele heredar todas las operaciones y atributos de su antecesor. El nuevo objeto puede definir nuevos atributos, nuevas operaciones, y redefinir las operaciones heredadas si se considera necesario.

Otro de los componentes importantes de una metodología orientada a objetos es la comunicación entre objetos. Esta comunicación se lleva a cabo cuando un objeto invoca una operación de otro objeto. Al invocar la operación el objeto invocante puede pasar parámetros de entrada al objeto invocado. Al finalizar la operación, se pueden devolver parámetros de salida.

# Modelado mediante UML

**El lenguaje unificado de modelado (UML) permite representar el sistema mediante diagramas a niveles diferentes:**

- ***modelo de objetos*: mediante diagramas de objetos que representan las clases de objetos y su jerarquía**
- ***modelo dinámico*: mediante diagramas de estados que representan el comportamiento dinámico de cada objeto**
- ***modelo funcional*: mediante diagramas de flujo de datos que expresan las entradas, salidas y funciones de cada objeto**

**En la fase de análisis no se entra en los detalles informáticos; debe ser entendible por el usuario.**



## Notas:

El lenguaje Unificado de MOdelado (UML) permite la descripción de un modelo de un sistema software mediante diagramas que permiten describir diversos aspectos.

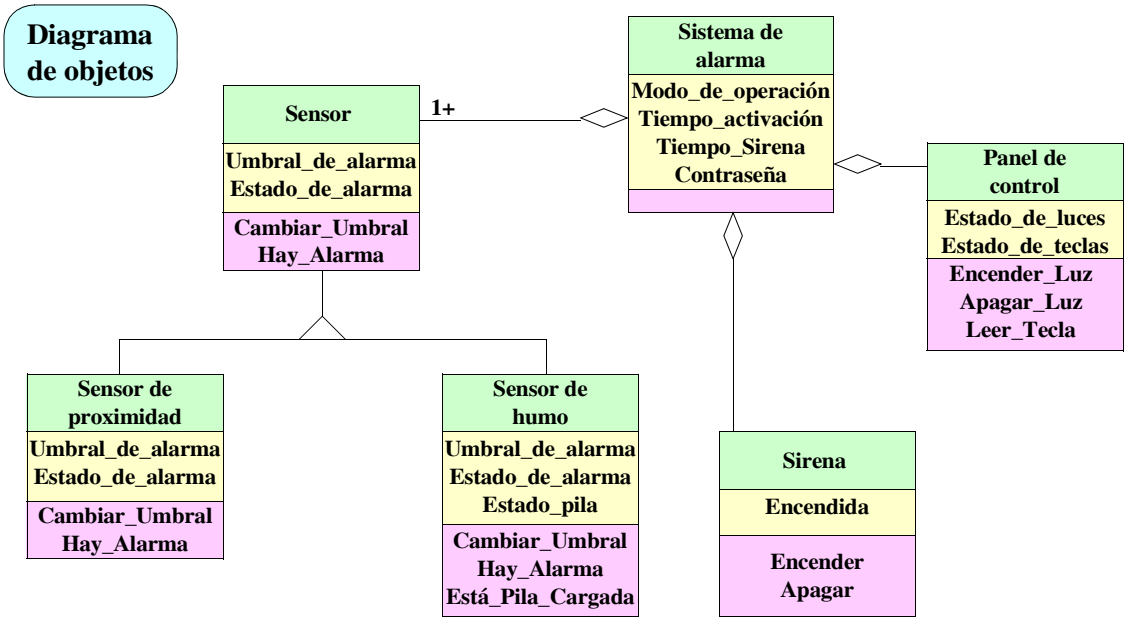
El sistema se descompone en objetos y se modela desde varios puntos de vista diferentes (de forma parecida a los planos de un edificio, que se representan desde diferentes puntos de vista):

- **Modelo de objetos**: mediante diagramas de objetos que presentan las clases de objetos y los objetos. Cada objeto se representa con su nombre, sus atributos y sus operaciones. Se muestra asimismo la jerarquía entre objetos y las relaciones que hay entre ellos.
- **Modelo dinámico**: mediante diagramas de estados que representan los estados en los que puede estar cada objeto, y las transiciones entre estos estados, causadas por la activación de eventos.
- **Modelo funcional**: mediante diagramas de flujo de datos que representan para cada objeto las entradas, salidas, y funciones que realizan.

El documento final de análisis se compone del planteamiento inicial del problema (requerimientos), más los modelos de objetos, dinámico y funcional.

En ocasiones la frontera entre el análisis y diseño no está clara. Como norma general, el análisis debe ser entendible por el usuario, y el diseño contiene los detalles de la implementación.

# Ejemplo: Sistema de Alarma:






## Notas:

En el diagrama de objetos de la figura superior se muestra una descomposición de un sistema de alarma en clases de objetos. Como puede verse, la clase sensor se ha extendido para dar lugar a una clase sensor de proximidad y una clase sensor de humos. Ambas heredan los atributos y operaciones de su antecesor y, el sensor de humos, añade además una operación para conocer el estado de carga de las pilas. La herencia se muestra en el diagrama por medio de un pequeño triángulo sobre una barra horizontal, con líneas verticales que indican la relación de parentesco.

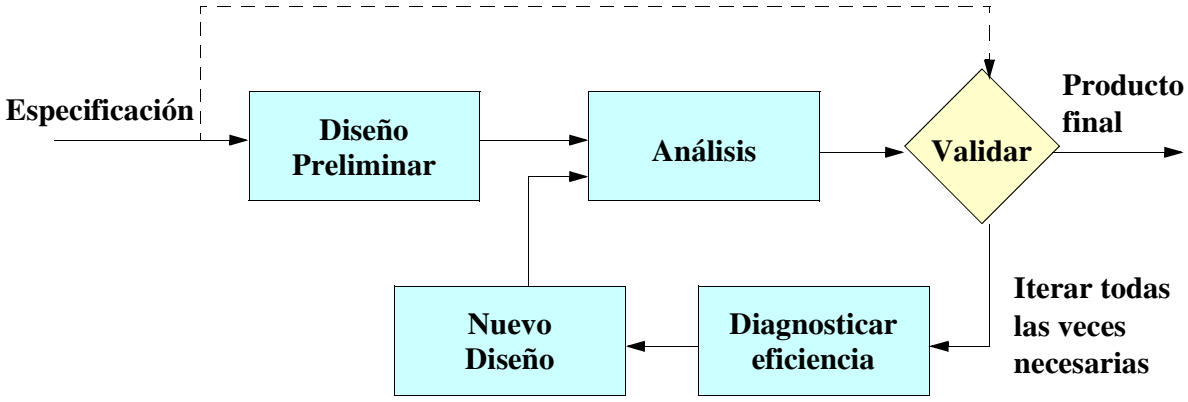
El resto de las clases de objetos no tiene herencia, aunque la posterior descomposición interna que se realice puede traer la definición de nuevas clases y, quizás la herencia. Por ejemplo, el panel de control tiene varias luces, y varias teclas, que se podrían especificar cada uno como un objeto individual; las teclas se podrían especializar después en teclas luminosas y no luminosas.

Las líneas que aparecen uniendo diferentes objetos representan las relaciones entre objetos. Las marcadas mediante un rombo representan la agregación, es decir indican que un objeto se compone de otros objetos más elementales. Así el sistema de alarma, que representa al sistema total, es una agregación de sí mismo más uno o más sensores, mas un panel de control, y una sirena.

A este diagrama habría que añadirle el modelo dinámico y el modelo funcional, que se expresan de forma similar a lo visto para el análisis estructurado.

# 3. Técnicas de diseño

## La naturaleza del proceso de diseño es altamente iterativa, y se basa en los principios de abstracción, refinamiento sucesivo, modularidad, y ocultamiento de información






## Notas:

Las técnicas de diseño de programas han de tener en cuenta la naturaleza del proceso de diseño, que generalmente es altamente iterativo.

Normalmente el diseño se realiza en varias etapas, y se progresa por refinamientos sucesivos. Primero se realiza un diseño preliminar, de alto nivel. Posteriormente se van realizando otros diseños, más detallados, y con diferentes niveles de abstracción.

El diseño se hace siempre modular, con objeto de dividir el programa en trozos de tamaño manejable. Luego, cada trozo o módulo se puede descomponer en nuevos módulos.

Al dividir un sistema en módulos se expresan sus componentes externas, y se ocultan los detalles internos. Sólo a la hora de diseñar el módulo concreto, se presta atención a los detalles internos. Esta separación de los detalles internos y de los elementos visibles desde fuera se denomina ocultamiento de información.

# Etapas del diseño

## Generalmente suele hacerse el diseño en dos etapas:

- Diseño de la arquitectura
  - diseño de las estructuras de datos
  - estructura general del programa
  - interfaces entre los diferentes módulos
- Diseño detallado
  - diseño detallado de las estructuras de datos
  - diseño de los algoritmos o procedimental



## Notas:

Aunque el diseño se realiza en varias etapas, por refinamientos sucesivos, suele dividirse en dos grandes bloques: el diseño de la arquitectura, y el diseño detallado.

En el diseño de la arquitectura se divide el sistema en módulos y se expresan las funciones de cada módulo, sus operaciones, y las interfaces necesarias para usar el módulo desde el exterior. Asimismo se diseñan las estructuras de datos que se van a utilizar, definiendo los módulos en que se van a situar, y sus interfaces (pero no su estructura interna). La arquitectura del sistema se puede expresar con varios niveles de abstracción.

Una vez finalizado el diseño de la arquitectura comienza el diseño detallado, en el que se especifican para cada módulo los algoritmos que se van a utilizar para cada operación o procedimiento, y las estructuras de datos concretas que se van a utilizar para el almacenamiento de información.

A menudo, el diseño detallado se va realizando en paralelo a la codificación. Sin embargo es necesario que el diseño de la arquitectura se haga antes de escribir el software, para garantizar la calidad del producto final.

# Algunas técnicas de diseño

## Diseño de la arquitectura:

- diseño modular
- diseño descendente y ascendente
- diseño estructurado
- diseño orientado a objetos

## Diseño detallado:

- programación estructurada
- programación orientada al objeto
- programación redundante
- programación defensiva



## Notas:

En la transparencia superior se muestran algunas de las técnicas de diseño que se utilizan más habitualmente en el diseño de la arquitectura y el diseño detallado de un programa. Estas técnicas de diseño no son excluyentes, sino que aparecen generalmente unidas, en los diferentes niveles de diseño.

- *diseño modular*: descomposición del programa en módulos independientes
- *diseño descendente y ascendente*: la partición en módulos se puede realizar de lo general a lo más detallado (descendente), o de lo más detallado a lo general (ascendente)
- *diseño estructurado*: asociado al análisis estructurado
- *diseño orientado a objetos*: asociado al análisis orientado a objetos
- *programación estructurada*: técnicas de programación que evitan los saltos incondicionales (Go To)
- *programación orientada al objeto*: ayudan a programar diseños orientados a objetos
- *programación redundante*: para detectar y corregir fallos (tolerancia a fallos)
- *programación defensiva*: para detectar todos los fallos que sea posible y garantizar un mejor funcionamiento del programa.

# Diseño modular

## Un módulo realiza una función específica e independiente

## Debe ser autocontenido y evitar los acoplos:

- Acoplo por datos.
- Acoplo por control. Paso de nombres de módulos o variables de control entre módulos.
- Acoplo por estructuras de datos globales declaradas como externas.
- Acoplo por estructuras de datos globales.
- Acoplo por contenido. Un módulo hace un salto al interior de otro.



## Notas:

Un módulo de un programa se puede definir como un subprograma de unas dimensiones no muy grandes, que realiza una función específica e independiente.

Un módulo es por naturaleza autocontenido. Ello significa que su eliminación de un sistema solamente deshabilita la función única realizada por ese módulo. Igualmente, si sustituimos el módulo por una versión más actualizada de éste, sin modificaciones sobre las variables de entrada y salida, el sistema debe funcionar básicamente igual.

Cuando diseñamos software basado en módulos que operan según estos principios, el diseño es denominado modular.

La principal dificultad de la programación modular consiste en evitar el acoplo entre los módulos (hacerlos independientes).

Como guía para los distintos acoplos posibles, de mejor a peor, puede utilizarse la de MYERS (1978), que se muestra en la transparencia superior.

# Características de la programación modular

## Más fácil escribir el programa

## Proceso de dirección más sencillo

## Facilita el ocultamiento de información y la abstracción

## Requiere esfuerzo y cuidado en el diseño, para hacer los módulos independientes

## Modificación más sencilla

## Más fácil probar el programa



## Notas:

La programación modular presenta innumerables ventajas. En primer lugar, al permitir la descomposición del programa en trozos de tamaño manejable hace más fácil de escribir y probar el programa, y hace el proceso de dirección más sencillo.

Asimismo, la estructura de los módulos facilita el ocultamiento de información ya que se pueden especificar los detalles internos por un lado, y la parte visible por otro.

La programación modular facilita la abstracción, ya que se puede describir el sistema con varios niveles de abstracción, de forma jerárquica, con módulos compuestos a su vez por otros módulos.

Sin embargo, la programación modular requiere esfuerzo y cuidado en el diseño, para hacer los módulos independientes. Si los módulos no son independientes no se consigue una verdadera programación modular.

Si el programa es modular, su modificación es más sencilla, ya que la independencia entre los módulos facilita el que un cambio en un módulo no afecte al resto.

Además, el programa es más fácil de probar, ya que podemos hacer las pruebas de cada módulo por separado, y luego realizar la prueba del programa conjunto separadamente (prueba de integración). De esta forma es mucho más fácil localizar y corregir los errores, la hacerse la prueba paso a paso.

# Diseño ascendente y descendente

## Diseño Descendente o "top-down":

- La estructura de control del programa se fracciona en módulos
- El diseño de estos módulos principales se hace en primer lugar
- Los módulos se dividen en submódulos, de forma jerárquica
- Los detalles de diseño de módulos inferiores permanecen escondidos



## Notas:

El diseño **"top-down" o descendente** es un proceso de descomposición focalizado principalmente en el flujo de control o la estructura de control del programa.

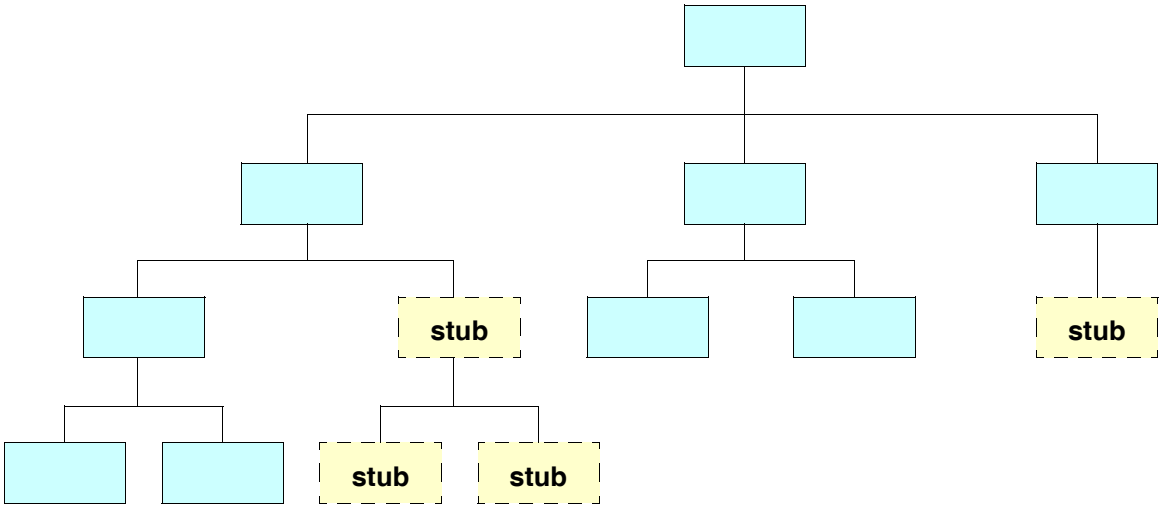
El primer paso es el estudio de todos los aspectos del programa y el fraccionamiento de éste en un número (3 a 10) de **módulos independientes**.

El segundo paso es la división de cada uno de estos módulos en submódulos independientes, repitiéndose el proceso hasta obtener módulos suficientemente pequeños como para su fácil retención mental, y así poder realizar su diseño detallado y codificación sin complicaciones.

Una de las características más importantes de esta técnica es que los detalles del diseño de los niveles inferiores aparecen **escondidos**. Solamente es necesario definir los datos y el control que actúan sobre las interfaces entre cada módulo.

Por tanto, la programación "Top-down" permite "olvidarse" de los detalles en los niveles superiores del diseño, aunque esto puede impedir darse cuenta a tiempo de que el camino elegido no es el correcto para alcanzar las especificaciones.

# Desarrollo de un diseño "top down"






## Notas:

En la figura superior se muestra un instante en el desarrollo de un programa diseñado mediante la técnica "top-down".

En este desarrollo, los módulos que aún no se han implementado se sustituyen por "stubs", o prototipos sencillos. Esto permite construir prototipos de un sistema incompleto, para probar parte de la funcionalidad, y que el usuario pueda validar el proceso.

# Técnica de diseño ascendente

## Diseño ascendente o “bottom-up”:

- Primero se hace una planificación de los módulos de bajo nivel que se vayan a necesitar
- Se desarrollan las partes más detalladas y con mayor nivel de dificultad en primer lugar
- Se realiza el diseño del resto del sistema, acomodando los diseños previos, hasta llegar finalmente al diseño del sistema final.
- Puede resultar una estructura de control inadecuada



## Notas:

En un diseño **“bottom-up”, o ascendente,** primero se idea un diseño típico del sistema, por experiencia, intuición o un análisis rápido, y se decide qué partes del diseño son más difíciles o con mayores limitaciones.

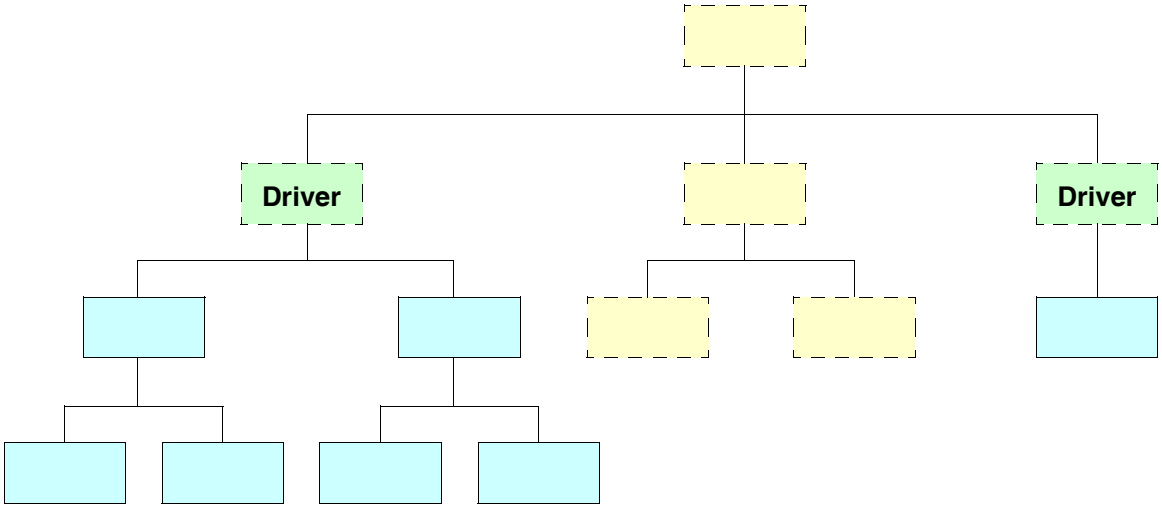
Estas partes esenciales se desarrollan en primer lugar, y el resto del diseño es confeccionado para acomodar los diseños, ya escogidos, de las partes esenciales.

En el diseño “bottom-up” el principal problema puede ser la acomodación de los diferentes módulos en una estructura de control inadecuada.

En muchas ocasiones se utilizan juntas las técnicas de programación “top-down” y “bottom-up” mezcladas, intentando aprovechar las mejores características de cada una.

El tipo de aproximación utilizada presenta una gran influencia no sólo en la fase de diseño, sino en la metodología de desarrollo y prueba del programa.

# Desarrollo de un diseño "bottom-up"



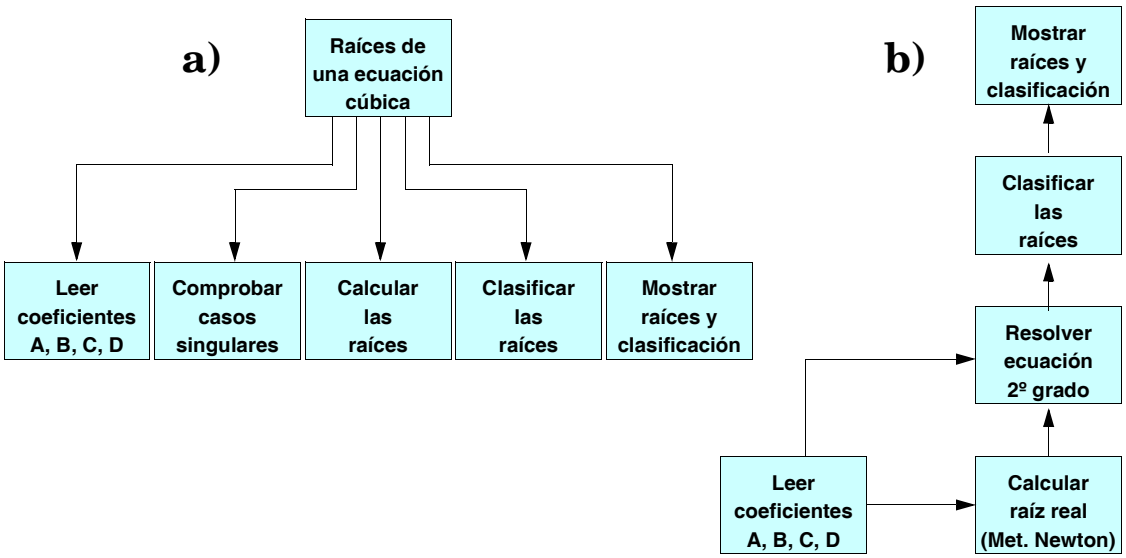



## Notas:

En la figura superior se muestra un instante en el desarrollo de un programa diseñado mediante la técnica "bottom-up".

En este desarrollo, los módulos que aún no se han implementado, que son los del nivel superior, se sustituyen por "drivers", o programas principales sencillos, cuya función es llamar a los módulos que están desarrollados, y permitir su prueba. Esto permite crear prototipos que ensayan parte de las funciones del sistema final, para su validación por parte del usuario.

# Ejemplo de programación "top-down" y "bottom-up"






## Notas:

Este ejemplo representa el diseño de un programa para obtener las raíces de una ecuación cúbica.

En el apartado (a), se realiza el diseño del módulo principal, y posteriormente se diseñan los módulos del nivel inferior.

Durante la implementación algunos de los módulos inferiores pueden ser sustituidos por prototipos, con los que se comunica el módulo principal.

En el apartado (b) se comienza por el módulo que representa el algoritmo esencial del problema: cálculo de una raíz real mediante el método de Newton. Posteriormente se procede con la fórmula cuadrática, etc., para finalizar con la salida de resultados.

Durante su implementación, se realizará la prueba del programa mediante un programa principal prototipo, que se comunica con los módulos que se van desarrollando.

# Diseño orientado a objetos

UNIVERSIDAD DE CANTABRIA

**Si el análisis es orientado a objetos, es lógico hacer diseño orientado a objeto**

**En el diseño de la arquitectura se realizan las siguientes etapas:**

- **se divide el sistema en subsistemas**
- **se identifica la concurrencia entre objetos**
- **se eligen las estrategias de almacenamiento de datos**
- **se elige la estrategia general de programación: máquinas de estados abstractas, tareas concurrentes, etc.**

**En el diseño detallado se desarrollan los detalles de cada objeto: operaciones, atributos, relaciones entre objetos, etc.**



## Notas:

UNIVERSIDAD DE CANTABRIA

Si el análisis del sistema es un análisis orientado a objeto, lo lógico es hacer el diseño también mediante una metodología orientada al objeto. En el análisis de la arquitectura se realizan las siguientes actividades:

- organizar el sistema en subsistemas
- identificar la concurrencia entre objetos
- asociar objetos a recursos (por ejemplo a procesadores)
- elegir la estrategia básica de almacenamiento de información (ficheros, sistemas distribuidos, datos en memoria, etc.)
- elegir la estrategia general de implementación, que puede ser una máquina de estados abstracta, tareas concurrentes, etc.
- considerar los aspectos del entorno
- establecer prioridades

En el diseño detallado lo que se pretende es una definición detallada de cada objeto. Generalmente se hacen diagramas más detallados que en el análisis tanto para los objetos, como para su comportamiento dinámico y funcional. Para cada objeto se eligen los algoritmos para sus operaciones, las representaciones concretas de cada atributo, y la forma de implementar las diferentes relaciones entre los objetos.

# Programación orientada a objetos

UNIVERSIDAD DE CANTABRIA

**Existen lenguajes de programación orientada a objetos que facilitan la implementación de un diseño orientado a objetos.**

**Los principales mecanismos son:**

- **encapsulamiento**
- **herencia**
- **polimorfismo**

**También es posible implementar un diseño orientado a objetos mediante lenguajes normales**

- **pero se repite mucho código y no se aprovechan todas las ventajas del diseño**



## Notas:

UNIVERSIDAD DE CANTABRIA

La programación orientada al objeto se halla facilitada en los lenguajes de programación orientada al objeto, que soportan los siguientes mecanismos:

- Encapsulamiento de los objetos o clases de objetos junto a su estado y sus operaciones, con el ocultamiento de todos los detalles internos del objeto.
- Creación de nuevos objetos a partir de otros existentes por medio de la herencia de todos sus atributos y operaciones.
- El polimorfismo, que permite invocar a una operación de una determinada categoría de objetos si saber expresamente el tipo concreto del objeto que se va a usar. En el instante de la ejecución, cuando ya se sabe el tipo concreto del objeto, se ejecuta la operación adecuada a ese objeto de forma automática.

La programación orientada al objeto para determinados tipos de sistemas también se facilita si el lenguaje permite la ejecución concurrente de múltiples objetos, mediante la utilización de técnicas de multitarea. Esto permite la creación de objetos o clases de objetos activos.

# Programación redundante

## Muchos de los sistemas basados en computador deben de ser tolerantes a fallos:

- controladores de vehículos
- controladores de sistemas críticos (centrales nucleares, control de tráfico aéreo, etc.)
- bases de datos (bancos, etc.)

## La tolerancia a fallos se consigue mediante la redundancia:

- *física*: varios computadores haciendo lo mismo
- *lógica*: varios algoritmos haciendo los mismos cálculos de forma diferente

## Esta última es la programación redundante



## Notas:

Existen muchos sistemas que deben de construirse de forma que sean tolerantes a fallos. Típicamente los controladores de vehículos son así (coches, aviones, satélites, etc.). También hay sistemas de seguridad crítica que requieren tolerancia a fallos, como controladores de centrales nucleares o del tráfico aéreo. Asimismo algunas aplicaciones de bases de datos contienen datos muy valiosos y deben funcionar aunque haya algún fallo (por ejemplo, las bases de datos de los bancos)

La tolerancia a fallos se consigue replicando diversas partes del equipo:

- *Redundancia física*. Permite evitar los efectos de errores del hardware. Para ello se replican la CPU, los discos, memoria, etc. Generalmente es preciso tener un sistema de detección de los fallos para que, si son repetitivos, se sustituya la unidad que funciona mal mientras las demás siguen realizando su labor.
- *Redundancia lógica*. Permite evitar fallos del software. Consiste en repetir los mismos algoritmos pero escritos de forma distinta. Para ello se suelen desarrollar los algoritmos por dos equipos totalmente diferentes, para que la probabilidad de que tengan el mismo fallo sea muy baja. También es necesario un sistema que sincronice los resultados del cálculo y determine los errores que puedan darse para desechar esos resultados. Si los resultados son muy diferentes se puede utilizar el que más se acerque a los valores esperados.

# Programación defensiva

## Hay que suponer siempre que va a haber errores

## Hay que comprobar:

- datos de entrada de periféricos (rango y atributos)
- datos proporcionados por otros programas
- consistencia de las estructuras de datos
- entradas de operadores (naturaleza, secuencia,...)
- límites de stacks
- límites de arrays
- posibilidad de división por cero u otras operaciones aritméticas incorrectas en las expresiones.
- salidas hacia otros programas o periféricos



## Notas:

Una de las estrategias más importantes para generar software seguro es la programación defensiva, que puede ser activa o pasiva.

La programación defensiva pasiva comprueba la información en determinados puntos del programa, para detectar la existencia de errores.

La programación defensiva activa opera de forma periódica o durante períodos de actividad baja del computador analizando la información almacenada para verificar la presencia o no de errores.

En general la programación defensiva requiere que el programador tenga presente que la aparición de errores es muy probable, y por tanto debe de introducir los elementos de comprobación que considere necesarios.

Los elementos de información que típicamente deben de comprobarse en programación defensiva son los que se indican en la transparencia de arriba.