Software:
Tecnología para el Procesamiento de Información

Jorge L. Ortega Arjona
Departamento de Matemáticas
Facultad de Ciencias, UNAM

Introducción

- La humanidad controla **la materia y la energía** mediante ciencia y tecnología.
- En contraste, a la fecha el **procesamiento de información** no había tenido modificación o cambio considerable, debido tal vez a que el cerebro humano es un poderoso medio para el manejo y control de información.

Introducción

Desarrollos anteriores:

- El **lenguaje escrito** (5 mil años aprox.) es la capacidad de registrar información trascendiendo espacio y tiempo: almacenar, recobrar y comunicar información
- Las **operaciones aritméticas simples y la representación numérica** (4 mil años, aprox.) son la habilidad de manipular datos cuantitativos
- Los **métodos de impresión** (500 años) es la creación de copias idénticas del mismo registro, para difundirlo a un número mayor de personas

Introducción

Software: un Nuevo Desarrollo

- Hoy, mucho de la actividad humana depende del **procesamiento de información**
- La información puede **ser almacenada, recobrada, comunicada, reordenada, seleccionada, dirigida y transformada y difundida** en grandes cantidades y velocidades usando **software**
- Todo procesamiento mecánico y repetitivo de información puede hacerse usando computadoras y **software**
- Cualquier procesamiento, en forma algorítmica (**una secuencia de operaciones que pueda ser precisamente especificada**), puede realizarse sin mayor intervención humana

Pero ¿qué es Software?

- Ingeniería de Software
- Arquitectura de Software
- Software Propietario
- Software de Aplicación
- Software Base
- Utilerías de Software
- Herramientas de Software
- Software de Control de Sistemas
- Etc…

Sí, pero ¿qué es Software?

Algo que

- Se programa..
- Se desarrolla...
- Se diseña...
- Se compra...
- Se copia...
- Se corre, se ejecuta...
- Se administra...
- Se modela...
- Etc...

¿Qué es Software?

- **Software** es el conjunto de instrucciones que le dicen a la computadora qué hacer
- Actualmente su **importancia es mayor** que la computadora misma: una computadora sin software es tan sólo un máquina inútil compuesta de circuitos electrónicos
- La **cantidad de conocimiento** necesario para crear el software básico que convierta a la máquina en una computadora útil es **comparable** al requerido para **crear la computadora** misma
- El proceso de creación de software, conocido como **programación**, puede ser el alfabetismo del tercer milenio: el conocimiento de software será parte importante de la educación

Programación: Lenguajes de Alto nivel

- Un lenguaje de alto nivel, o algebraico, permite una **representación** del **software** en términos de lenguaje natural humano.
- Lenguajes de alto nivel: Ada, Algol, APL, Basic, C, Cobol, Fortran, Lisp, Pascal, Simula, C++, Java, Haskel, Miranda.

Programación: Lenguajes de Alto nivel

- **Cada instrucción se compone o traduce en un cierto número de instrucciones más sencillas en lenguaje de máquina**
- **La mayoría puede realizar las mismas tareas básicas**
- Se requiere un **software traductor** que convierta un programa fuente en lenguaje de alto nivel a lenguaje de máquina: **intérprete ó compilador**.

Programación: Intérpretes y Compiladores

- **Un intérprete trabaja durante el tiempo de ejecución:** empezando con la primera instrucción, va traduciendo una a una las instrucciones a lenguaje de máquina, **ejecutándolas enseguida**
- **Un compilador traduce todo el programa fuente a un programa objeto en lenguaje de máquina.** El programa se traduce fuera de tiempo de ejecución. El programa objeto se almacena, y **se ejecuta al invocarlo o cargarlo en memoria**.

Programación: Estructuras de Datos

- Cualquier lenguaje de alto nivel puede usar **estructuras de datos** (formas de organizar datos en memoria: arreglos, listas, cadenas, árboles, colas y pilas)
- La eficiencia en la operación sobre los datos depende de la **representación** de las estructuras de datos, y la manera como organiza su **almacenamiento** en memoria

Algunas preguntas sobre Programación

En mis clases, pregunto a mis estudiantes:

- Primera pregunta: **¿Quién sabe programar?**
- Segunda pregunta: **¿Quién sabe programar bien?**
- Tercera pregunta: **¿Cómo saben que programan bien?**

¿Cómo enseñamos y aprendemos a programar?

¿Cómo enseñamos y aprendemos a programar?

Revisando el temario de diversos cursos de programación:

- Se enseña programación desde hace mucho tiempo mediante visualizar un programa como un **producto terminado**, con todas sus líneas de código ordenadas correctamente, su sintáxis del lenguaje correcta, y su apariencia muy clara.
- El objetivo es que el programa **sea aceptado por el compilador**, y luego averiguar **si realiza la función requerida** con las propiedades adecuadas.
- **Solo aquellos con "el don" son capaces de programar bien.**

¿Cómo enseñamos y aprendemos a programar?

Aprender a programar **es una experiencia personal**:

- En la licenciatura tomé dos cursos de programación; estos cursos **se limitaban a describir un lenguaje y pedir pequeños ejercicios** de resolución de problemas. Terminé programando en **ensamblador del 68000.**
- En la maestría, di el salto a la Programación Orientada a Objetos y Concurrencia. **Se esperaba que yo ya tuviera un manejo ágil de programación y lenguajes de programación.**
- En conclusión: **aprendí a programar en general como un proceso de prueba y error**, y no como un progreso estructurado y ordenado. De nuevo, la tercera pregunta: **¿Cómo saber si yo programo bien?**

¿Cómo aprender a programar bien?

Una propuesta:

- Se requiere un cambio de visión sobre la programación, de **"un producto"** a **"un desarrollo"**.
- Esto coloca el énfasis donde debe estar: en **sucesivamente re-escribir y re-pensar** que va moldeando el acto de programar en la mejor manera posible.
- **Si el desarrollo es sólido, el producto también lo será.**

¿Cómo aprender a programar bien?

Una propuesta:

- Programar, como desarrollo, **organiza y clarifica el pensamiento**. Escribir programas es como vamos pensando el camino a una solución, que nos vamos apropiando paso a paso.

- No es suficiente escribir líneas de código claro; es necesario organizar esas líneas en **una forma coherente**, hasta obtener un resultado.

¿Cómo aprender a programar bien?

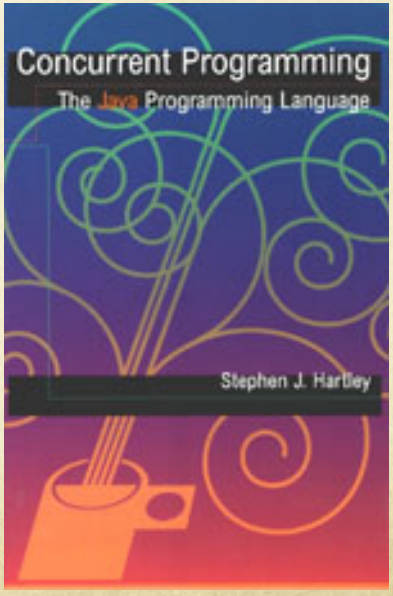
Una propuesta:

- Programar bien no puede enseñarse en el vacío. Se puede aprender a programar siguiendo buenos "ejemplos".
- Debemos enseñar a nuestros estudiantes: "esto es como otros han programado en esta área, y funciona porque cumple con tal y cual característica; **léelo, estúdialo, y piensa en ello; tú también puedes hacerlo**".

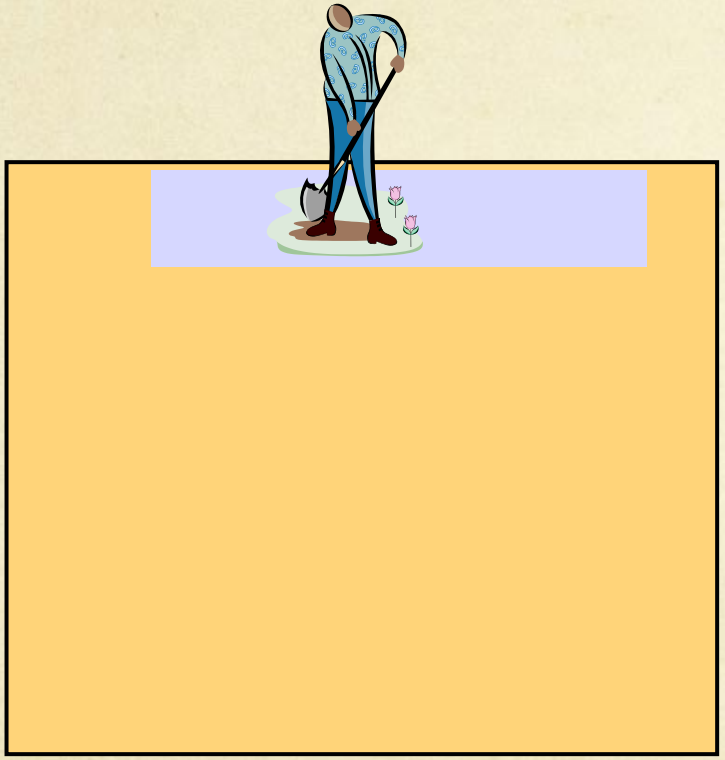
¿Cómo aprender a programar bien?

En mis clases de Programación Concurrente:

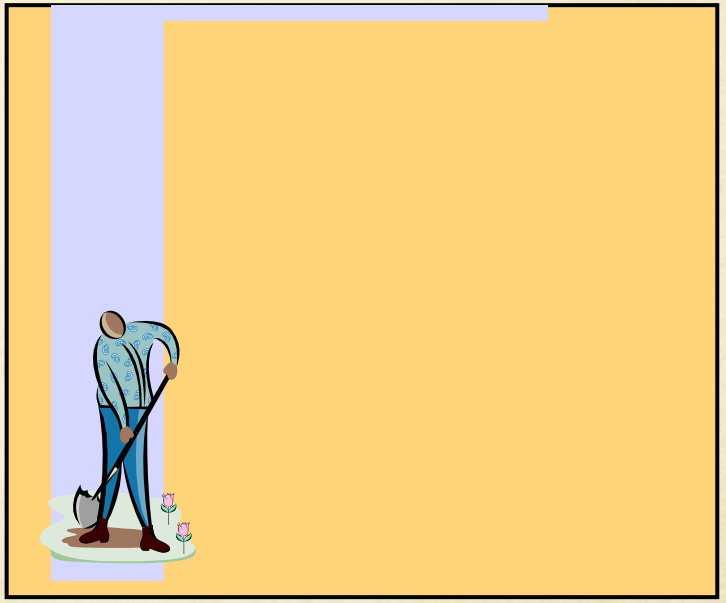
- Programación Concurrente es un tema **difícil de programar.**
- Encontré el libro "*Concurrent Programming. The Java Programming Language*", de Stephen Hartley (1998).
  - Cubre **todos los temas** que yo estudié sobre Programación Concurrente.
  - Todos los ejemplos compilan **sin errores ni advertencias**.
  - Está lleno de ejemplos cuya programación es clara; pido a mis estudiantes que no solo lo compilen y lo ejecuten: **léanlo, estúdienlo, y piensen en ellos.**



En mi opinión:
¿Cómo deberíamos aprender a programar?



Cómo aprendemos a programar



Cómo deberíamos aprender a programar

Programar se basa en escribir

- La base de toda la teoría sobre programación de software es la **escritura**; los lenguajes de programación son solo símiles de la escritura.
- **Programar bien se basa en escribir bien; escribir bien se basa en pensar bien.**
- Para obtener software con ciertas características, debemos estudiar **cómo tales carácterísticas se encuentran en el propio software**.

Successful Software Engineering Research

David L. Parnas (1998)

- Investigación en otras áreas de Ingeniería
- ¿Qué paradigma se sigue en la investigación en Ingeniería de Software?
- **¿Quién está estudiando métodos de inspección?**
- **¿Quién está tomando una seria visión de la documentación?**
- Conclusiones

Successful Software Engineering Research

David L. Parnas (1998)

En investigación en otras áreas de Ingeniería:

- Los **problemas** de diseño o producto se encuentran en la **práctica real**; se usan **abstracciones** para identificar **características fundamentales**, y el análisis provee de **conocimiento** sobre tales características.
- Después de investigar modelos abstractos, el investigador provee una **contribución original** sobre cómo resolver el problema de diseño o producto para los desarrolladores. **Tal contribución es el objetivo de la investigación**.
- La literatura de investigación cuenta con muchos problemas, y la contribución **se añade a tal literatura**.

Successful Software Engineering Research

David L. Parnas (1998)

¿Qué paradigma se sigue en la investigación en Ingeniería de Software?

- Examinando la literatura en "Ingeniería de Software", **los artículos siguen paradigmas de otras áreas** que no son de Ingeniería. Algunos árbitros de revistas juzgan artículos **con estándares de áreas que no son de Ingeniería**.

- La mayoría de desarrolladores de software **ignoran la mayoría de la investigación en Ingeniería de Software**, pues **no parece resolver cuestiones que interesan a los desarrolladores, u ofrecen soluciones que puedan utilizar.**

Successful Software Engineering Research

David L. Parnas (1998)

¿Qué paradigma se sigue en la investigación en Ingeniería de Software? (cont.)

- Los desarrolladores de software no buscan en la literatura de investigación, sino en **revistas populares de software** que ofrecen **descripciones superficiales y respuestas fáciles**.
- La Ingeniería en general, y la Ingeniería de Software en particular, **es siempre difícil**. Las presiones del mercado nos forza a ser mejores, pero las respuestas fáciles son generalmente no-respuestas en absoluto; **las respuestas fáciles son distracciones**.
- **"No encuentro contribuciones sólidas y útiles en la mayoría de las revistas populares de software."**

Successful Software Engineering Research

David L. Parnas (1998)

¿Quién está estudiando métodos de inspección?

- La **inspección de software es un problema mayor** en muchos ambientes de desarrollo. La industria requiere métodos que ayuden a proceder sistemáticamente, considerando cuidadosamente todos los casos **para proveer confianza que nada ha sido pasado por alto**.

- Las principales propuestas no vienen de la investigación académica, sino de practicantes o consultores**. Estas publicaciones pragmáticas se enfocan en aspectos administrativos/organizacionales**, sin tomar en cuenta el vasto cuerpo de literatura e investigación en métodos matemáticos de verificación.

Successful Software Engineering Research

David L. Parnas (1998)

¿Quién está echando una mirada seria a la documentación?

- Mucho tiempo y dinero se desperdicia por la **pobre calidad de la documentación** de software de mantenimiento.
- "Pregunta a tu desarrollador de software favorito porqué se hizo un error, y muy probablemente te dirá que **la documentación no era clara, completa, consistente, o precisa**".
- **Los programas son muy precisos y sensibles a cambios menores.**
- Documentación completa debe incluir **mucho detalle** y cubrir muchos casos diferentes.

Successful Software Engineering Research

David L. Parnas (1998)

¿Quién está echando una mirada seria a la documentación? (cont.)

- Hay descripciones sobre **qué hacen los programas, no cómo lo hacen**; tienden a ser voluminosas y deben organizarse de forma que *(a)* la información que necesitas sea **fácil de encontrar**, y *(b)* faltantes e inexactitudes puedan ser **detectadas**.

- Encontrar formas para documentar programas que se organice alrededor de la **recuperación de información** es un campo fértil para investigadores en Ingeniería de Software que busquen tener impacto en el desarrollo de software.

Successful Software Engineering Research

David L. Parnas (1998)

¿Quién está echando una mirada seria a la documentación? (cont.)

- La educación en Ingeniería muestra **cómo las matemáticas juegan un papel esencial en la documentación** de productos de ingeniería. Sin embargo, mucha de la literatura en **"métodos formales" no ofrecen soluciones** para el problema de la documentación de software. **Las especificaciones son difíciles de escribir, pero son más difíciles de leer.** En muchos casos, se espera que el lector derive el comportamiento de un sistema sutil de axiomas.

- En matemáticas para ingeniería, el comportamiento en la documentación se describe por fórmulas que se evalùan mediante substituir valores.

Successful Software Engineering Research

David L. Parnas (1998)

Conclusiones

- Manténgase atento a lo que está pasando en la realidad **mediante leer programas industriales**.
- Trate de **aplicar sus ideas a los programas que se han escrito para otro propósito**, no a programas que ha hecho para ilustrar sus ideas.
- No ataque los síntomas, pero **manténgase buscando las causas**. Los desarrolladores podrán y atacarán los síntomas tan bien como puedan.
- Manténgase preguntando **porqué la gente no está usando sus ideas**, y no tome "estupidez" o "ignorancia" como respuestas. No puede eliminar la estupidez y puede hacer poco para corregir la ignorancia, pero si hay una debilidad en los resultados existentes de investigación, ha encontrado un problema de investigación sólido.

Successful Software Engineering Research

David L. Parnas (1998)

Conclusiones

- **Tenga cuidado con las modas**. Durante mi carrera he visto muchos tópicos que se han hecho muy populares. ¿Quién hoy está seriamente interesado en Algol 68, PL/1 ó Ada? Sin embargo no hace mucho la literatura estaba llena de artículos en esos tópicos. La investigación es muy dada a las modas en campos donde cada artículo es una respuesta a otro, en lugar de una respuesta a un problema fundamental. **Siempre busque el problema fundamental y no salte al carro de la moda**.

- **Tenga cuidado con** *buzzwords* **vagamente definidas**. Un *buzzword* es una palabra que todo el mundo sabe, pero pocos pueden definir. "*Buzzword*" es un *buzzword*. Mucha de la literatura actual es un debate sobre el significado de las palabras disfrazado de un debate acerca de cómo diseñar software. Por ejemplo, muchos de los debates acerca de fortalezas y debilidades de la OO se reducen a qué significa OO.

Successful Software Engineering Research

David L. Parnas (1998)

Conclusiones

"The secret to successful research is picking the right problem. I have known many people who were better at solving problems that I am, but I have been honoured by SIGSOFT's award because I found my research problems by working with developers."

"El secreto para una investigación exitosa es escoger el problema correcto. He conocido mucha gente que es mejor que yo para resolver problemas, pero se me honra con el premio SIGSOFT porque he encontrado mis problemas de investigación mediante trabajar con desarrolladores".

No Silver Bullet

Essence and Accidents of Software Engineering

Fred Brooks Jr. (1987)

- "*Fashioning complex conceptual constructs is the essence; accidental tasks arise in representing the constructs in language. Past progress has so reduced the accidental tasks that future progress now depends upon addressing the essence.*"

- "La elaboración de construcciones conceptuales y complejas es la **esencia**; las tareas **accidentales** surgen al interpretar construcciones en un lenguaje. El progreso en el pasado ha reducido tanto las tareas accidentales que el progreso futuro depende ahora en abordar la esencia."

No Silver Bullet

Essence and Accidents of Software Engineering

Fred Brooks Jr. (1987)

- **Dificultades Esenciales:** inherentes de la naturaleza del software
- "La esencia de una entidad de software es una construcción de conceptos interconectados: conjuntos de datos, algoritmos, e invocaciones a funciones. Esta esencia es abstracta dado que la construcción conceptual es la misma bajo diferentes representaciones"
- **Dificultades accidentales:** atienden a la producción de software, pero no son inherentes.

"I believe the hard part of building software to be the specification, design and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation"

No Silver Bullet

Essence and Accidents of Software Engineering

Fred Brooks Jr. (1987)

Dificultades Esenciales

- **Complejidad**. No hay dos partes iguales (a nivel instrucción); gran número de estados; no escalable linealmente.
- **Conformidad.** Interfaces entre instituciones humanas a las que debe adecuarse. La variación en interfaces no se debe a necesidad, más bien porque han sido diseñados por diferentes personas.
- **Cambiabilidad.** El software cambia constante debido a que incorpora funcionalidad y porque puede ser "fácilmente" cambiado.
- **Invisibilidad**. El software es invisible y no visualizable. Las abstracciones geométricas no capturan su esencia, ya que el software se encuentra inherentemente inmerso en la memoria.

No Silver Bullet

Essence and Accidents of Software Engineering

Fred Brooks Jr. (1987)

Logros que resolvieron Dificultades Accidentales

- **Lenguajes de Alto nivel.** Liberan al software de su complejidad accidental, dando al programa abstracciones manejables
- **Tiempo Compartido.** Preserva la inmediación, lo que permite mantener en mente las complejidades del software
- **Ambientes unificados de programación.** Atacan la dificultad accidental de pensar individualmente en programas

Esperanzas: Ada y otros lenguajes de alto nivel, **Programación Orientada a Objetos**, Inteligencia Artificial, Sistemas Expertos, Programación "Automática", Programación Gráfica, Verificación de Programas, Ambientes y Herramientas, Estaciones de Trabajo

¿Qué es Software, de nuevo?

Software:

Es el “material” del que las funciones de procesamiento se construye

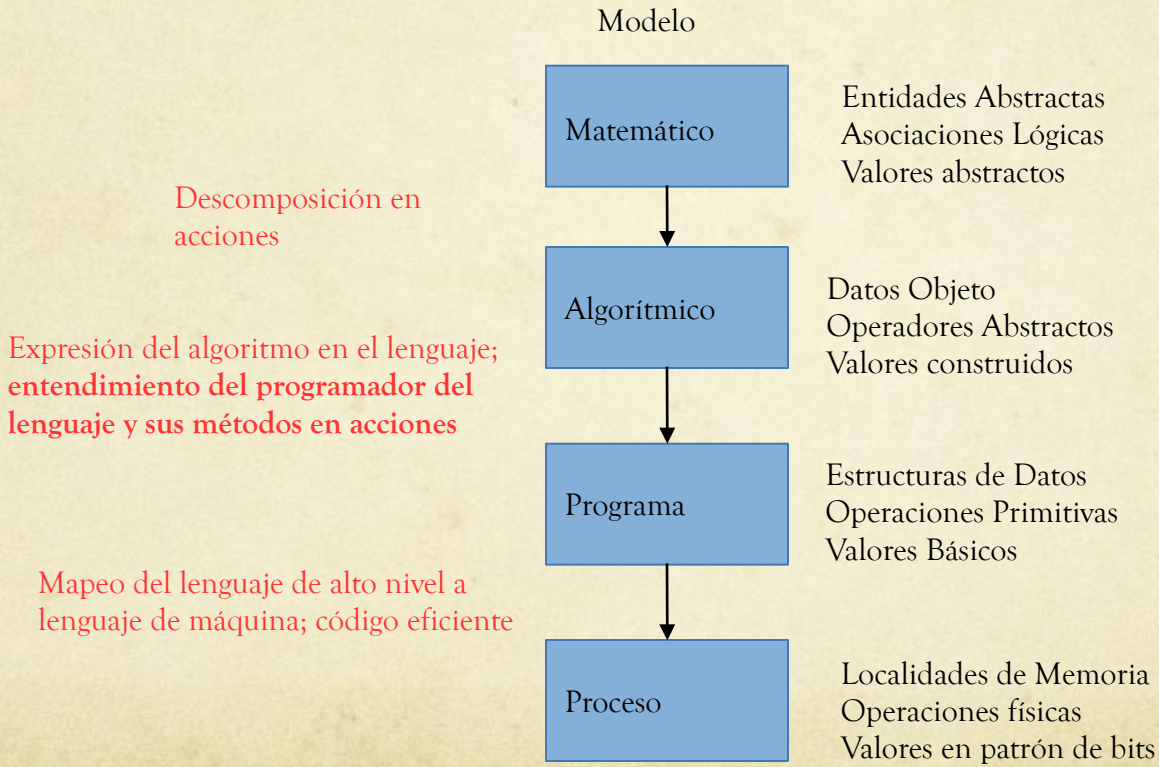
Es algo usado en la computadora, pero es más que un autómata programado

Es un aspecto clave de cómo los programadores organizan una función en términos de una forma, mediante composición

Es más que código neutral: las preguntas son **¿cómo puede organizarse? ¿qué forma tiene esa organización?**

El Concepto de Software como Proceso

Proceso: Cambio en el estado de la memoria por acción del procesador



*C.M. Pancake & D. Bergmark (1990)

Sistema de Software (Procesos)

Sistema de software: colección, integración o ensamble de (un gran número de) partes o componentes independientes de software

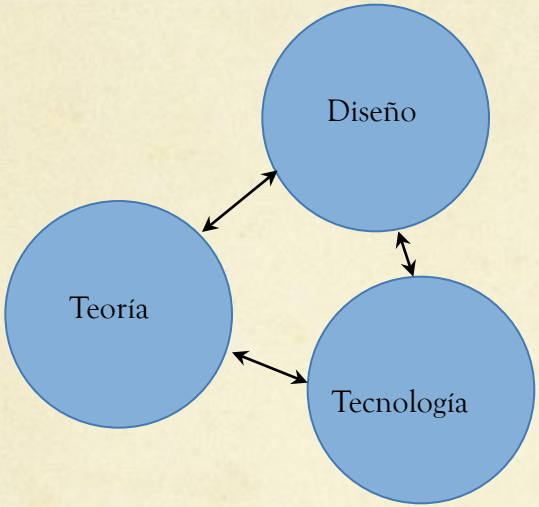
- **Complejo:** compuesto de partes interconectadas o entremezcladas

Arquitectura de Software

Arquitectura de software: disciplina o estudio de la descripción de sistemas de software como resultado del ensamblado de componentes de software

- **Abstracción:** eliminar detalles irrelevantes en la descripción de un sistema

Arquitectura de Software



Diseño de software: estudio de los principios fundamentales y técnicas de composición para crear sistemas de software, y sus descripciones formales e informales

Teoría de software: conjunto de conceptos y términos usados para definir las partes e interacciones de los sistemas de software

Tecnología de software: descripciones de cómo los sistemas de software se implementan, y las tecnologías relacionadas tanto en hardware como software

Diseño de Software

Debido a su complejidad, el Software es un reto para la aplicación de conocimiento de Diseño

El Software:

Provee una funcionalidad extendible por otras piezas de software, que representan otras funcionalidades

Es una entidad técnica y económicamente evolucionable

No es una entidad independiente, sino parte de un sistema de hardware y software

Diseño de Software en la Ingeniería de Software (1960s)

En Ingeniería de Software tradicional, el Diseño de Software es visto como un proceso formal de definir especificaciones y derivar un sistema de software de ellas.

El resultado del diseño es **un producto** (artefacto, máquina, sistema).

El producto **se deriva de las especificaciones dadas por el cliente**. En principio, con suficiente conocimiento y poder de cómputo, tal derivación puede automatizarse.

Una vez que el cliente y diseñador han acordado las especificaciones, **hay poca necesidad de contacto entre ellos** hasta la entrega del producto.

Diseño de Software centrado en el Humano (1980s)

En la rama de Diseño de Software centrado en el Humano, el Diseño de Software es entender el dominio de trabajo en el cual las personas interactúan con computadoras, realizando programación para facilitar la actividad humana.

El resultado de un buen diseño es **un cliente satisfecho**.

El proceso de diseño es **una colaboración entre diseñadores y clientes**. El diseño evoluciona y se adapta a sus preocupaciones cambiantes.

El proceso produce **una especificación como producto intermedio importante**.

Cliente y diseñador **están en constante comunicación durante todo el proceso**.

Una crítica sobre estas aproximaciones de Diseño de Software

El proceso de Diseño de Software propuesto por la Ingeniería de Software, al enfocarse solo en el sistema de software y sus especificaciones, **pierde de vista las necesidades comunes y acciones de las personas involucradas en su uso**.

Como está constituido, el Diseño centrado en el Humano **carece de formalismos y es incapaz de conectar sistemáticamente entre las necesiadades del usuario y la estructura del software**.

Por tanto, una disciplina de Diseño de Software debe **entrenar a quienes la practican para ser observadores expertos del dominio de acción** en el cual una comunidad de personas labora, de tal manera que el diseñador pueda **producir software que asista a las personas de la comunidad** para realizar efectivamente sus acciones**.**

Diseño y Arquitectura de Software

El Diseño de Software es una **práctica inherentemente complicada e irregular**, pero la Arquitectura de Software permite organizarlo en forma de un **proceso con carácterísticas establecidas e identificables**.

Diseño y Arquitectura de Software

El Diseño de Software es, predominantemente, una **mezcla ecléctica entre lo racional y lo intuitivo**. Desde el punto de vista de la Arquitectura, puede considerarse como **el proceso de desarrollo intuitivo de conceptos racionales**, basado en actividades:

- Análisis de alcance (*scoping*)
- Partición
- Agregación
- Integración
- Certificación

Estas actividades se repiten una y otra vez a lo largo de todo el proceso.

Diseño y Arquitectura de Software

El Diseño de Software gira alrededor de la **creación de documentos.** En Arquitectura de Software, tales documentos **describen modelos como resultado de cada paso del proceso**.

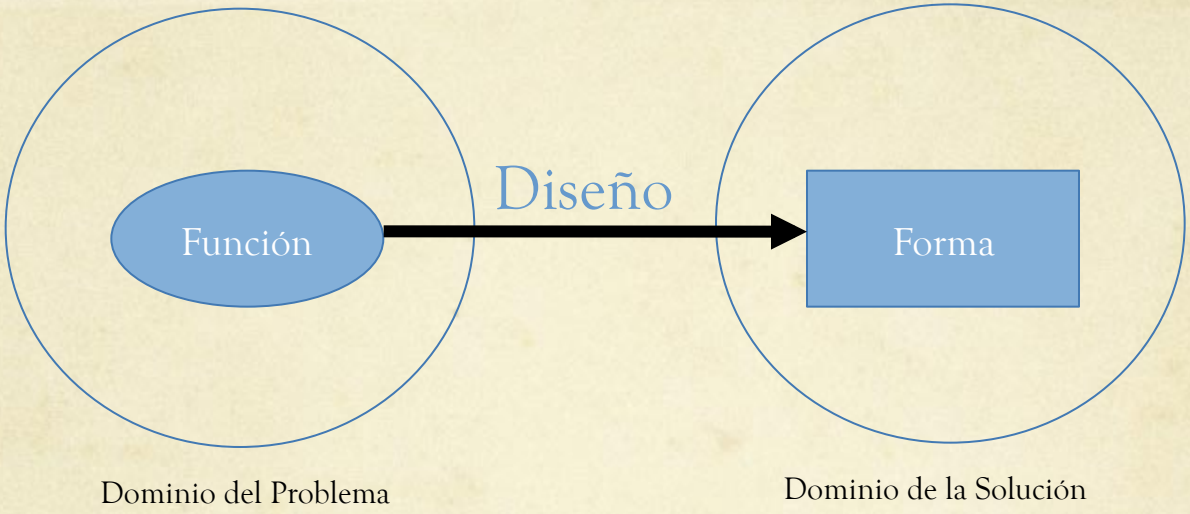
- Modelos de objetivo y propósito
- Modelos de forma
- Modelos de comportamiento (función)
- Modelos de desempeño
- Modelos de datos
- Modelos de administración

Diseño y Arquitectura de Software

Como el diseño de cualquier sistema complejo, **el Diseño de Software es inherentemente incierto.**

Al comenzar, en general es **poco claro** cuál será el resultado final. La Arquitectura de Software organiza al Diseño de Software como una **progresión contínua, dirigida a reducir o controlar (pero no eliminar) la incertidumbre.**

Diseño en general y Diseño de Software



Diseño es encontrar una forma que satisfaga a la función y a sus requerimientos. Es una actividad que obtiene una forma a partir de la descripción de una función.

En Software, forma y función pueden verse como organizaciones. Una organización de la función describe al software a nivel comportamiento; una organización de la forma describe al software a nivel estructura

Diseño de Software es una relación de una clase de organizaciones posibles de la función y una clase organizaciones posibles de la forma

Diseño en general y Diseño de Software

Diseño de Software se realiza mediante:

(a) organizar los componentes y conexiones de software para formar un sistema

(b) representar al propio sistema de software

(c) organizarse dentro de un método

El problema se describe **como una función y cómo debe realizarse**, es decir sus requerimientos: un algoritmo y datos

La solución se describe como **una forma y sus propiedades**: componentes de software y relaciones o conexiones entre ellos

Idealmente, **requerimientos y propiedades se mapean entre sí**

La situación en Diseño de Software

El software solo puede accederse mediante descripciones como código fuente o código ejecutable, lo que lo hace no-visible

¿Qué tiene el diseño de otros artefactos que le hace falta al software?

- Una base de experiencia y técnicas de diseño de software
- Una representación tangible del producto, particularmente su estructura
- Mediciones y evaluaciones para determinar si los requerimientos deseados se encuentran como propiedades en el producto final

Una base de experiencia y técnicas

Industrias maduras tienden a generar **manuales de experiencia y técnicas de diseño**, describiendo **soluciones exitosas a problemas conocidos**

Diseñadores e ingenieros rara vez comienzan sus diseños desde cero, sino que **reusan soluciones de diseño conocidas con un registro de éxitos**

Un Patrón es **una descripción** que relaciona una **solución (forma)** que resuleve un **problema (función)** general de diseño, que ocurre en un **contexto** particular.

- Inicialmente propuestos para arquitectura de edificios (*A Pattern Language*, **Christopher Alexander**, 1977)

- Basados en la observación de ciertas **estructuras y temas recurrentes y perdurables**

En tal sentido, los **Patrones de Software** y la **Comunidad de Patrones** representan el más claro intento de colectar experiencia y técnicas de Diseño de Software

Arquitectura de Software y Patrones de Software

El proceso de Diseño de Software puede desarrollarse entre dos extremos:

1. Desde un **diseño intuitivo, creativo, "inspirado"**,

2. Hasta un **diseño formal, racional y sistemático**

El paso de uno a otro se logra a través de una larga experiencia, durante la cual los diseñadores eventualmente desarrollan y manejan patrones comprobados de función y forma.

Un Patrón de Software es **una relación forma-función** que ocurren en un **contexto** en el cual la función se describe en términos del dominio del **problema** como un conjunto de fuerzas, y la forma es una estructura descrita en términos del dominio de la **solución** que logra un equilibrio aceptable entre las fuerzas

Patrones de Software

Un Patrón de Software se refiere a una **configuración o forma de software** que realiza **una función de manera específica**, y que se realcionan entre sí dentro de un **contexto dado**.

Un Patrón de Software se describe mediante los siguientes puntos:

1. Nombre y resumen
2. **Contexto (“fuerzas”)**
3. **Problema**
4. **Solución (estructura, participantes, dinámica e implementación)**
5. Discusión de la resolución de fuerzas (nuevo contexto)
6. Descripción de cómo se relaciona elpatrón con otros patrones.

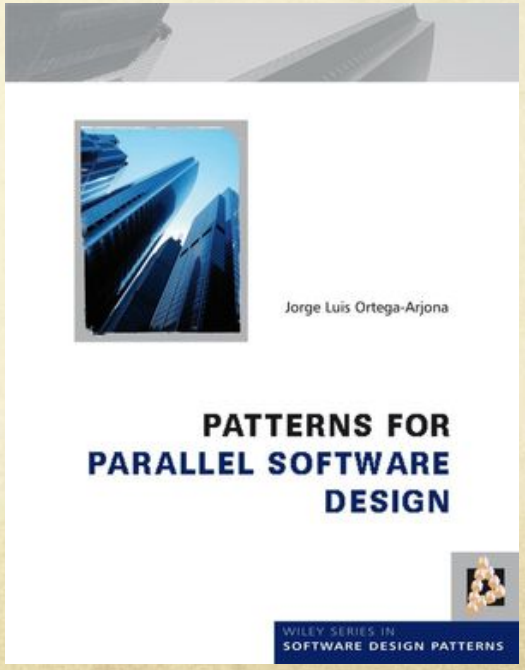
Patterns for Parallel Software Design

Jorge L. Ortega-Arjona (2010)

Los **Patrones para Diseño de Software Paralelo** son un conjunto de patrones de software que desarrollan la coordinación, la comunicación y la sincronización de un sistema de software paralelo.

Método de diseño:

1. Análisis del Problema
2. Diseño de la Coordinación
3. Diseño de la Comunicación
4. Diseño Detallado
5. Implementación y pruebas



Una descripción tangible

Una descripción tangible del software normalmente consiste de los componentes de software, las conexiones de software y una frontera del sistema de software

Los componentes de software se hacen tangibles mediante encerrarlos en contenedores de implementación

Todo nivel de contenedores de implementación tienen una relación de contención entre sí

How Buildings Learn?
What happens after they're built.

Steward Brand (1994)

La idea de las Capas de Cambio en un edificio surge de la forma en que el edificio es planeado y creado, siguiendo las necesidades actuales y futuras de sus ocupantes.

Capas de Cambio (en Arquitectura de Edificios):

1. "Site is eternal"
2. "Structure persists and dominates"
3. "Skin is mutable"
4. "Services obsolesce and wear out"
5. "Space plan is the stage of human comedy. New scene, new set"
6. "Stuff just keeps moving"

The Layers of Change in Software Architecture

Ortega-Arjona & Roberts (1999)

Software Site

- Tiene por objetivo proveer de una **base estable** sobre la cual construimos programas de software.
- Esto puede representarse simplemente **por los elementos de** *hardware* **de la computadora y el ambiente de software (sistema operativo)** en el que se desarrolla la programación.
- Una buena parte del desarrollo y construcción del software depende de estos elementos.

"Site is eternal"

The Layers of Change in Software Architecture

Ortega-Arjona & Roberts (1999)

Software Structure

- Tiene por objetivo **proveer estabilidad y apoyo** a las otras capas subsecuentes. Se representa por el esquema de organización básica del software.
- Es una descripción del software como un **conjunto de sub-sistemas, especificando sus responsabilidades, e incluyendo reglas y guías para organizar las relaciones entre ellos**.
- Responde a las cuestiones de particionar un sistema de software complejo.

"Structure persists and dominates"

The Layers of Change in Software Architecture

Ortega-Arjona & Roberts (1999)

Software Skin

- Tiene por objetivo **dar apariencia** al software, exhibiendo sus funcionalidades.
- Se representa por todos los componentes que permiten interacción con el usuario, principalmente **representadas por las interfaces gráficas de usuario**, en las que los usuarios finales conceptualizan el funcionamiento del software.
- Van desde los ambientes gráficos completos hasta las pantallas de texto simple.

"Skin is mutable"

The Layers of Change in Software Architecture

Ortega-Arjona & Roberts (1999)

Software Services

- Tiene por objetivo proveer apoyo para actividades comunes durante el uso del software.
- Son elementos relacionados con las "entrañas funcionales" del software: puede encontrarse como **aquellos componentes estándar preconstruidos que proveen funcionalidades comunes**, como bibliotecas matemáticas, de entrada/salida, o acceso a disco.
- Frecuentemente, deben ser ajustadas para su uso en un software específico.

"Services obsolesce and wear out"

The Layers of Change in Software Architecture

Ortega-Arjona & Roberts (1999)

Software Space plan

- Tiene por objetivo organizar las diferentes tareas parciales o actividades realizadas por el software.
- Se representa por la manera en que **estructuras de datos y funciones se organizan como abstracciones, en la forma de sub-sistemas, componentes e interfaces**.
- Por ejemplo, OO presenta organizaciones de clases que pueden ser usadas como plano de objetos cooperativos.

"Space plan is the stage of human comedy. New scene, new set"

The Layers of Change in Software Architecture

Ortega-Arjona & Roberts (1999)

Software Stuff

- Tiene por objetivo representar los elementos reales del software que realizan proceso o contienen información.
- Se representa por **funciones, procedimientos, representación de datos, estructuras de datos**, etc.

"Stuff just keeps moving"

La necesidad de mediciones

La principal razón para un aproximación de Diseño de Software es mejorar la calidad del software

"Calidad de software es el grado con que el software posee una combinación deseada de atributos"

Medición y evaluación. Habilidad de medir cualidades y evaluarlas para confirmar su presencia o ausencia en otros sistemas de software.

Repetitibilidad: los resultados pueden reproducirse en sistemas de software similares

Evaluación de alternativas. Como potenciales soluciones, los sistemas de software deben revisarse respecto a su habilidad de satisfacer la funcionalidad y los requerimientos

Designing Hard Software. The Essential Tasks

Bennet (1997)

La situación en Software

- "Esas metodologías y todo esa documentación de diseño solo se ponen en el camino del trabajo real" (un programador)
- "¿Has comenzado a programar aún?" (un gerente)
- "El equipo de desarrollo para un sistema que será implementado en una arquitectura de hardware cliente-servidor es responsable de todo, desde enunciar los casos anidados, para asegurar que el sistema logra las tasas de transacciones requeridas" (un plan de proyecto)
- "Puedo programarlo de cero más rápido que lo que puedo averiguar lo que hace el componente existente y modificarlo para que haga lo que necesito" (un programador)
- "No tenemos un arquitecto de sistemas, pero animamos a los grupos de desarrollo a hablar entre sí" (un administrador de proyecto)

Designing Hard Software. The Essential Tasks

Bennet (1997)

Mediciones y evauaciones

- Las mediciones son el elemento faltante que, cuando se desarrolle, permite **que las partes de una máquina sean reusables**.
- Los desarrolladores de Software luchan con la calidad y la reutilización por la falta de mediciones. **Tener objetivos completos y cuantificados de todas las propiedades deseadas en un sistema y mediciones en los documentos de diseño para corroborar que las propiedades están ahí**, es crítico para entregar sistemas de Software que expresen las propiedades.
- El problema es **que las mediciones actuales no proveen apoyo para el diseño de software**, pues no se basan **en propiedades del Software**, sino en requerimientos que quisieran que exhibiera.

Designing Hard Software. The Essential Tasks

Bennet (1997)

Mediciones y evauaciones

- Las propiedades del Software pueden clasificarse en **propiedades en tiempo de construcción** (*build-time properties*) y **propiedades en tiempo de ejecución** (*run-time properties*).

- Tal vez, la pregunta última que debemos tratar de responder en el futuro de las mediciones es **¿estamos ahora mejor que comparado con como lo hacíamos antes?** Solo es posible responder si se cuenta con mediciones que ayuden a observar propiedades del Sofware, verificando su inclusión como parte del diseño.

Una Invitación

- El mundo de la programación de software es fascinante, intrincado, gratificante, y a la vez inmisericorde, complicado y demandante de grandes esfuerzos y toneladas de paciencia
- Quisiera invitar a todo aquél con una computadora a su disposición a escribir un programa, o dos
- La sensación de logro es grandiosa cuando se observa en acción un programa que uno ha pensado, trabajado y construido
- Tal vez es porque el proceso es adictivo, y consume mucho más tiempo de lo que se cree en un principio
- Al final, lo cierto es que en cada programador hay un optimista, siempre pensando que cada error en el programa es el último.

Bibliografía

- S. Hartley. *Concurrent Programming. The Java Programming Language*. Oxford University Press, 1998.
- D.L. Parnas. *Successful Software Engineering Research.* ACM SIGSOFT Software Engineering Notes, 23(3), 64-68, 1998.
- Brooks, F. P. *No Silver Bullet. Essence and accidents of software engineering*. IEEE computer, 20(4), 10-19, 1987.
- Pancake, C. M., & Bergmark, D. *Do parallel languages respond to the needs of scientific programmers?* Computer, 23(12), 13-23, 1990.
- C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahal-King, & S. Angel. *A Pattern Language.* Oxford University Press, 1977.
- Ortega-Arjona, J. L. *Patterns for parallel software design*. John Wiley & Sons. 2010.
- Brand, Stewart. *How buildings learn: What happens after they're built*. Penguin, 1994.
- Ortega-Arjona, J. L. & Roberts G.. *The Layers of Change in Software Architecture*. First Working IFIP Conference on Software Architecture (WICSA'99). 1999.
- Bennett, Douglas W. *Designing Hard Software: the essential tasks*. Manning Publications Co., 1997.

¿Preguntas?