Tema 1
Introducción a la Ingeniería del Software

Software, calidad, procesos y métodos de desarrollo.

Contenidos

1. El software.
2. Ingeniería de software.
3. Procesos de Desarrollo.
4. El software en la actualidad.

01. El software

¿Qué es y para qué sirve el Software?

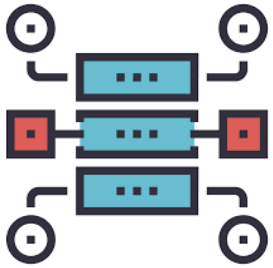
1. Software – ¿Qué es?

En la teoría…

El **software** es un conjunto de instrucciones y datos que le indican a una computadora cómo realizar cierta tarea de interés.

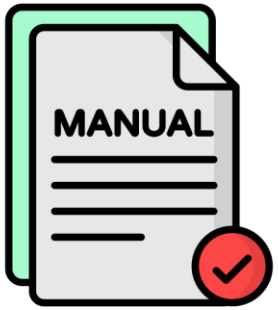
En la práctica…

Código



Datos

Persistencia



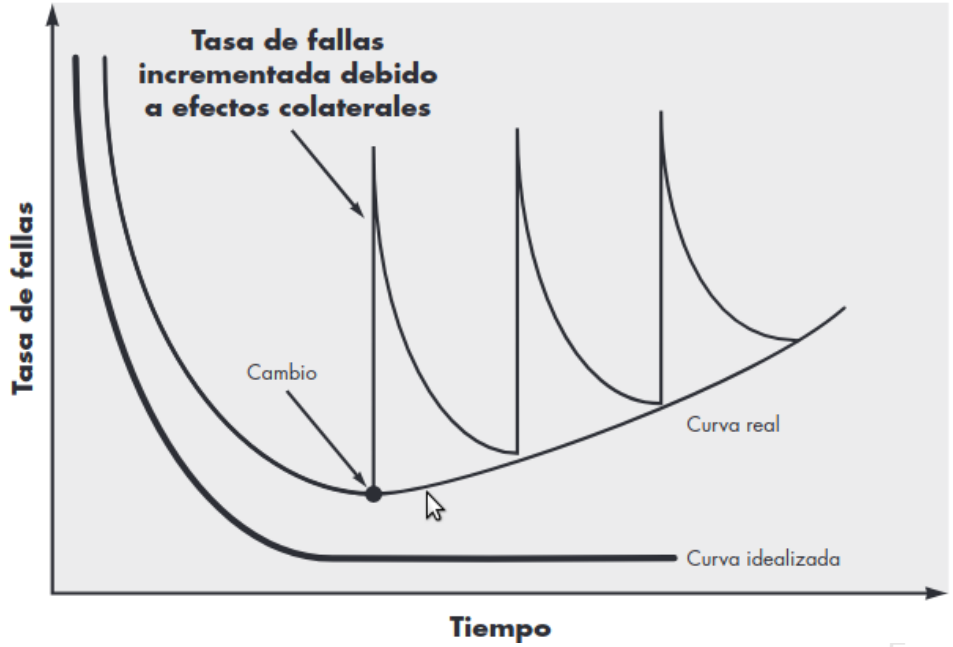
Documentación

Operación y mantenimiento

1. Software – ¿En qué se diferencia del hardware?

- El software es **lógico**, no físico.
  - Más difícil de medir, validar, verificar:
- El software se **desarrolla**, no se fabrica.
  - El coste está en el desarrollo, no en la copia.
- Se **construye a partir de componentes**.
  - Librerías, servicios, frameworks, …
- El software no se rompe, **se deteriora**.
  - Cambios
  - Deuda técnica
  - Entorno de ejecución

(Pressman, 2010)



1. Software – Problemas al desarrollar software

Retrasos y entregas fuera de plazo.

¿Por qué siempre se retrasa?

Sobrecostes y presupuestos irreales.

¿Por qué el software cuesta tanto?

Errores en producción, dificultad de mantenimiento.

¿Por qué llegan errores a producción?

Sensación de caos.

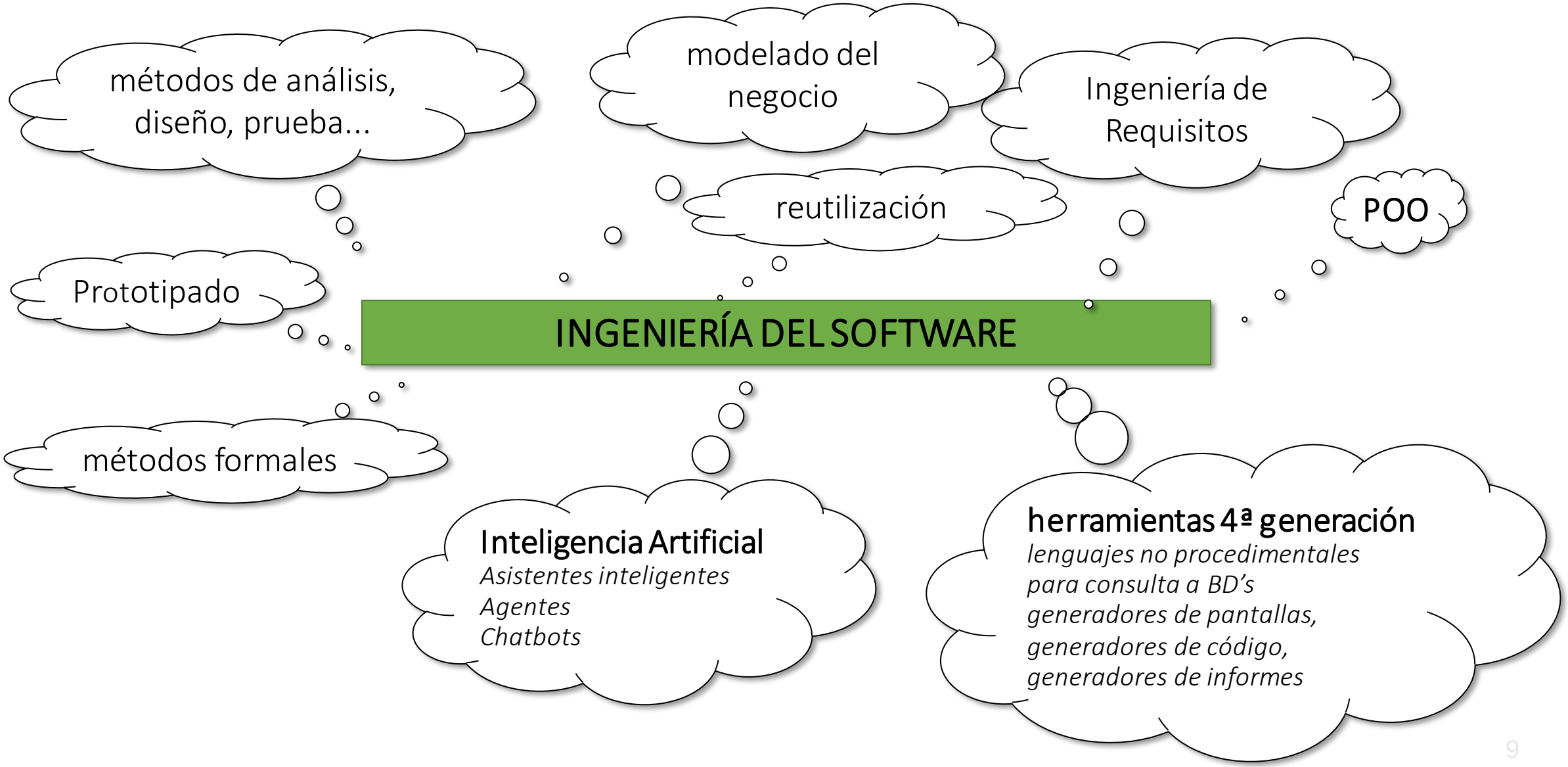
¿Por qué es tan difícil saber cómo vamos?

1. Software – Causas

Las causas de los problemas pueden clasificarse en:

- **Esenciales**
  Derivan de la propia naturaleza del software y del problema a resolver.
  Naturaleza lógica del software y alta maleabilidad (el software se puede cambiar fácilmente).
- **Accidentales**
  Derivan de las técnicas, herramientas y procesos.
  Problemas de comunicación con los clientes, poco esfuerzo en análisis y diseño, problemas de gestión de proyectos, ausencia de guías claras y datos históricos, ausencia de estándares ampliamente aceptados, ausencia de bases formales.

1. Software – Algunas soluciones



No Silver Bullet
—Essence and Accident in Software Engineering

Frederick P. Brooks, Jr.
University of North Carolina at Chapel Hill

There is no single development, in either technology or management technique, which by itself promises even one order-of-magnitude improvement within a decade in productivity, in reliability, in simplicity.

Abstract[1]

All software construction involves essential tasks, the fashioning of the complex conceptual structures that compose the abstract software entity, and accidental tasks, the representation of these abstract entities in programming languages and the mapping of these onto machine languages within space and speed constraints. Most of the big past gains in software productivity have come from removing artificial barriers that have made the accidental tasks inordinately hard, such as severe hardware constraints, awkward programming languages, lack of machine time. How much of what software engineers now do is still devoted to the accidental, as opposed to the essential? Unless it is more than 9/10 of all effort, shrinking all the accidental activities to zero time will not give an order of magnitude improvement.

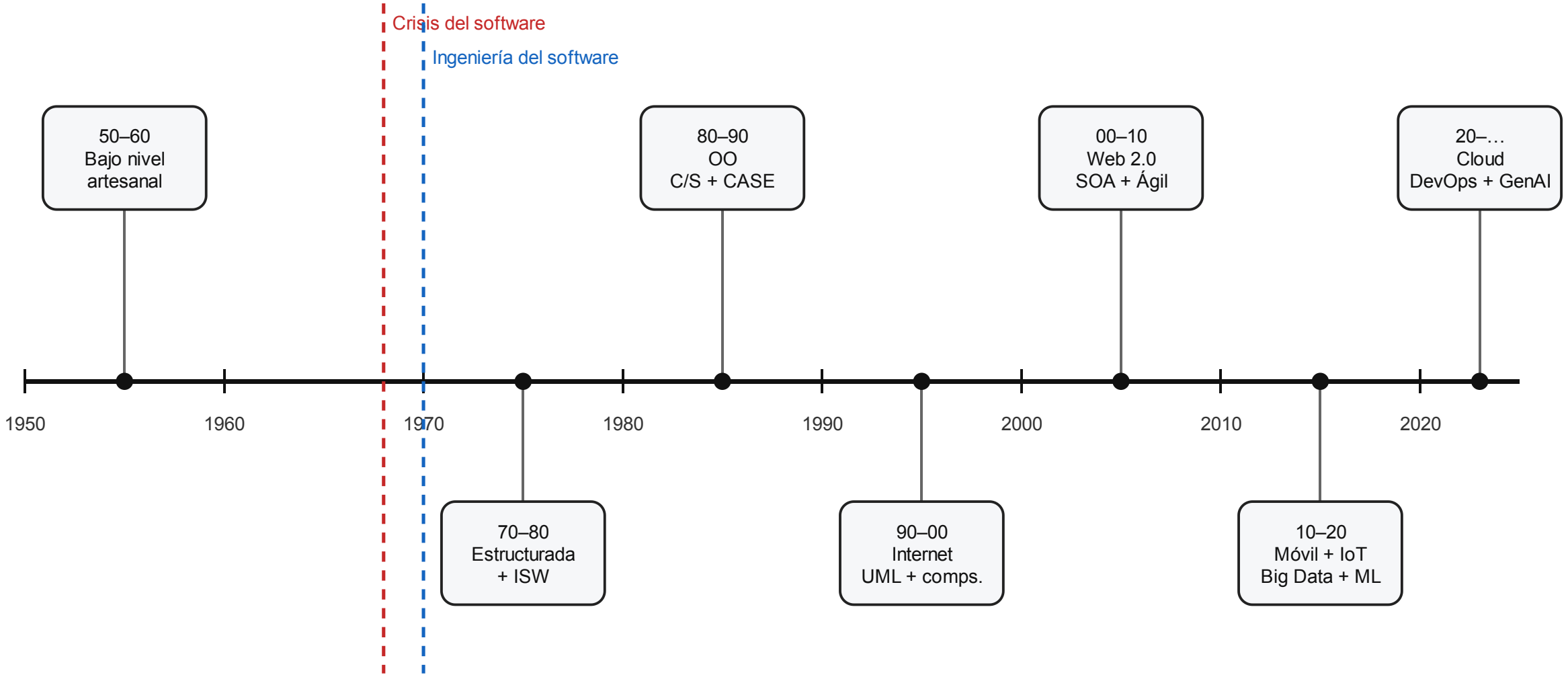
Therefore it appears that the time has come to address the essential parts of the software task, those concerned with fashioning abstract conceptual structures of great complexity. I suggest:

- Exploiting the mass market to avoid constructing what can be bought.
- Using rapid prototyping as part of a planned iteration in establishing software requirements.
- Growing software organically, adding more and more function to systems as they are run, used, and tested.
- Identifying and developing the great conceptual designers of the rising generation.

No existe una tecnología, método o herramienta que elimine de golpe los problemas del desarrollo de software.

No hay bala de plata

1. Software – Evolución



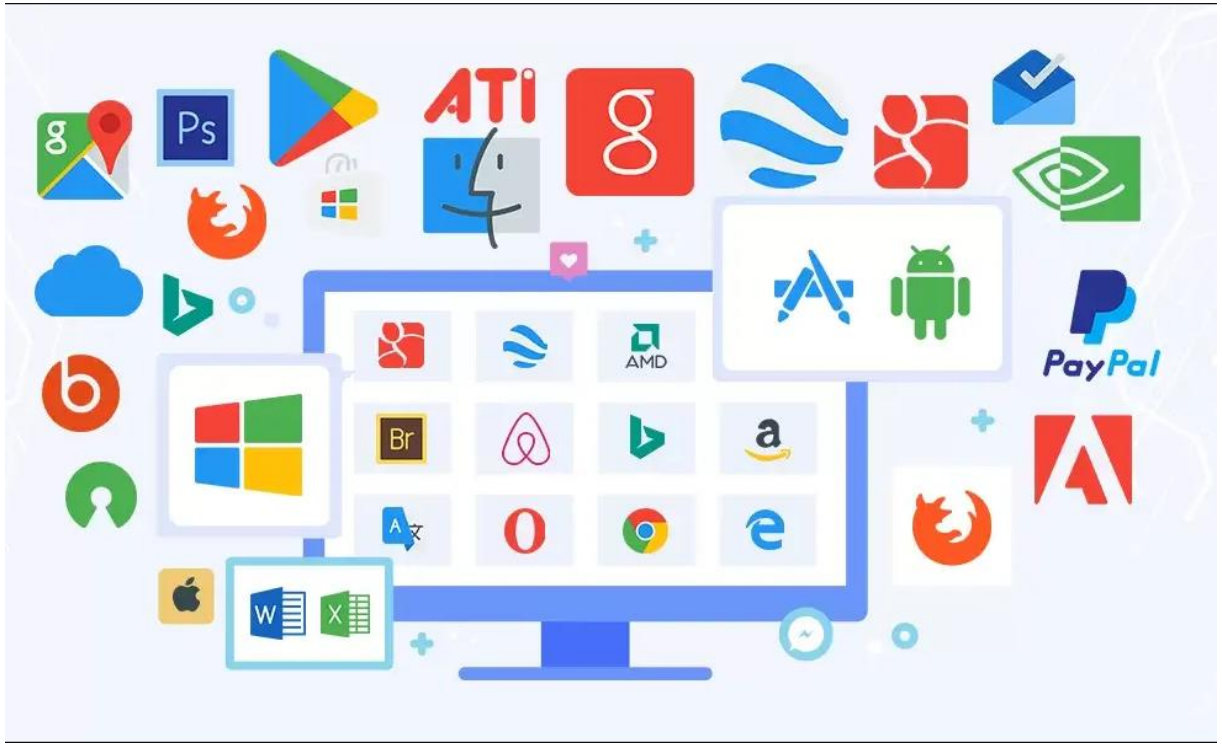
1. Software – Tipos de software

No todo el software es igual

Dependiendo del objetivo, la tecnología empleada, las expectativas, el estilo de los programas, o los requisitos, serán diferentes.

Ejemplos:

- Software de sistema
- Software de aplicación
- Software de desarrollo
- Software embebido
- Software científico
- Software de análisis de datos



2. Ingeniería del software

¿Por qué necesitamos la Ingeniería de Software?

2. Ingeniería de Software – Definición

La **ingeniería de software** es la aplicación sistemática de un proceso de diseño, desarrollo, prueba y mantenimiento de software. Es un enfoque sistemático y disciplinado para el desarrollo de software que tiene como objetivo crear software de alta calidad, confiable y fácil de mantener.

2. Ingeniería de Software – Otras definiciones

- (Bauer) “La ISW es el establecimiento y uso de **principios sólidos de ingeniería**, orientados a obtener **software económico** que sea **fiable** y trabaje de manera **eficiente** en máquinas reales”
- (IEEE 610, Glosario Estándar de Términos de Ingeniería del Software de IEEE) “ISW: (1) La aplicación de un **enfoque sistemático, disciplinado y cuantificable** para el desarrollo, la operación y el mantenimiento del software; es decir, la aplicación de la ingeniería al software; (2) El estudio de enfoques como en (1)”
- (Sommerville) “Disciplina que comprende todos los aspectos de la **producción de software** desde las etapas iniciales de la especificación del sistema, hasta el mantenimiento de éste después de que se utiliza”

2. Ingeniería de Software – Técnicas

- Modularidad.
- Abstracción.
- Encapsulación.
- Reutilización.
- Mantenimiento.
- Pruebas y verificación.
- Patrones de diseño.
- Metodologías ágiles.
- Integración y despliegue continuo.

2. Ingeniería de Software – Técnicas – Modularidad

La **modularidad** consiste en **dividir un sistema en módulos independientes**, cada uno con una responsabilidad bien definida.

- Un **módulo** expone **una interfaz clara y estable**.
- La implementación puede cambiarse sin afectar al resto del sistema.

Ventajas:

- División del trabajo entre equipos.
- Evolución y mantenimiento de partes aisladas del sistema.
- Mejor compresión del sistema.

2. Ingeniería de Software – Técnicas – Abstracción

La abstracción consiste en ocultar los detalles irrelevantes para centrarse en los aspectos esenciales de un problema.

¿Para qué se usa la abstracción?

- Reducir la complejidad.
- Facilitar la comprensión del sistema.
- Permitir trabajar a distintos niveles de detalle.

Ejemplos de abstracción

- **Funciones y procedimientos**: encapsulan tareas.
- **Clases y objetos**: Representan conceptos del dominio y su comportamiento.

2. Ingeniería de Software – Técnicas – Encapsulación

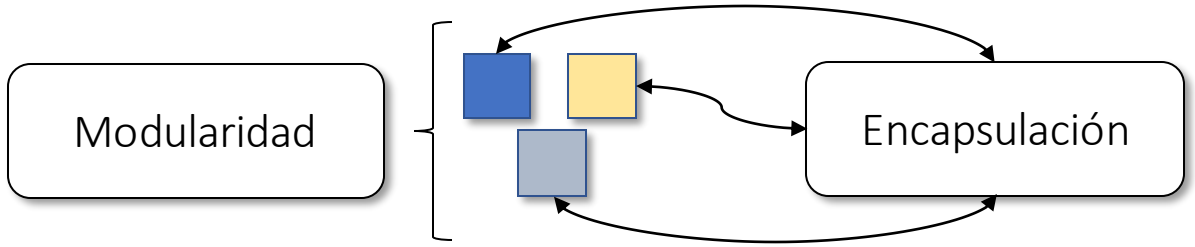
La encapsulación consiste en ocultar el estado interno y los detalles de implementación de un módulo, exponiendo únicamente una interfaz controlada para interactuar con él.

¿Para qué sirve?

- Proteger invariantes.
  (garantizar que el estado interno del módulo siempre sea válido)
- Reducir el acoplamiento entre módulos.
- Permitir cambios internos sin afectar al código cliente.

Relación con la modularidad

- La modularidad divide al sistema en partes.
- La encapsulación controla cómo se accede a esas partes.



2. Ingeniería de Software – Técnicas – Cohesión y acoplamiento

Cohesión
Grado en que los elementos de un módulo están relacionados.
Se busca una **única responsabilidad**.

Acoplamiento
Grado de dependencia del sistema.

Objetivo: +Cohesión –Acoplamiento

```
public class GestorDeUsuarios {
  public void crearUsuario() { }
  public void eliminarUsuario() { }
  public void enviarMensajeAUsuario() { }
  public void generarInforme() { }
  public void conectarBD() { }
}
```

```
public class ServicioPedidos {
  private RepoMySQL repo = new RepoMySQL();
  public void crearPedido() {
    // lógica para crear pedido
   repo.save();
  }
}
```

¿Qué problema tienen estos métodos?

2. Ingeniería de Software – Técnicas – Reutilización

La reutilización de software consiste en construir sistemas a partir de componentes existentes, en lugar de desarrollarlos desde cero.

Dos enfoques de reutilización.

- **Desarrollo con reutilización.**
  Uso de librerías, frameworks y componentes existentes.
- **Desarrollo para reutilización.**
  Diseño de componentes pensados para ser reutilizados en otros sistemas.

• Reutilización en la práctica.

- Ecosistemas de librerías y dependencias.
- Herramientas de construcción y gestión de dependencias (maven, pip, npm)
- Repositorios públicos de software reutilizable (PyPI, Maven central)

2. Ingeniería de Software – Técnicas – Reutilización

¿Cuándo construir o cuando reutilizar?

Reutilizar cuando:

- El problema es común y bien conocido.
- Existen librerías maduras y ampliamente usadas.
- El coste de desarrollar y validar la solución es alto.
- La dependencia es estable y bien mantenida.

Construir cuando:

- El problema es específico del dominio.
- Los requisitos son muy particulares.
- El control total sobre el código es crítico. https://en.wikipedia.org/wiki/Npm_left-pad_incident
- La dependencia introduce riesgos (versionado, seguridad, disponibilidad)

2. Ingeniería de Software – Técnicas – Patrones

Un **patrón de diseño** es una **solución reutilizable** para resolver cierto tipo de problema en el desarrollo de software.

- GoF (*Gang of Four*)
  https://refactoring.guru/es/design-patterns
- **Categorías**
  - **Creacionales.**
    Creación de objetos.
  - **Estructurales.**
    Organización y composición de clases y objetos.
  - **Comportamentales.**
    Comunicación y responsabilidades entre objetos.

Gang of Four (GoF)

- Ralph Johnson, Richard Helm, Erich Gamma, and John Vlissides (left to right)

2. Ingeniería de Software – Técnicas – Pruebas

Las **pruebas de software** son el proceso de ejecutar un programa con el objetivo de **detectar errores** y verificar su comportamiento esperado.

- En general, no es posible escribir un programa sin fallos al primer intento.
- Las pruebas **proporcionan confianza** *(no certeza absoluta)*.
- Permiten comprobar el comportamiento del software en diferentes escenarios.
- El diseño del software condiciona su capacidad de ser probado.
  - Un diseño acoplado o poco modular dificulta las pruebas.
- TDD (*Test-Driven Development*)
  Desarrollar el software escribiendo pruebas primero.

"Beware of bugs in the above code; I have only proved it correct, not tried it." - Donald Knuth

2. Ingeniería de Software – Técnicas – Análisis estático

El **análisis estático** es una técnica de verificación que examina el código sin ejecutarlo para detectar defectos, malas prácticas y vulnerabilidades.

- Busca problemas como:
  recursos no cerrados, código muerto, complejidad excesiva, posibles bugs, …
- Se apoya en **herramientas automáticas** (linters, analizadores, CI).

```
public void imprimirNombre(Usuario u) {
    System.out.println(u.getNombre().toUpperCase());
}
```

NullPointerException **in Java**

2. Ingeniería de Software – Técnicas – Frameworks

Un framework es una plataforma que proporciona una estructura, herramientas, librerías y convenciones para facilitar el desarrollo de aplicaciones de software.

Frameworks horizontales.

- Abordan una capa técnica o tipo de aplicación.
- Reutilizables en múltiples dominios.
- Ejemplos
  Spring Boot, React, Django.

Frameworks verticales.

- Abordan un dominio o industria concreta.
- Proporcionan sistemas completos pero personalizables.
- Ejemplos
  SAP, Salesforce, Magento

2. Ingeniería de Software – Tareas

El ingeniero de Software.

- Diseña y construye software más allá del código.
- Toma decisiones que afectan a Calidad, Coste, Tiempo y Mantenimiento.
- Trabaja con sistemas complejos, en equipo y a largo plazo.
- Aplica principios de ingeniería para gestionar la complejidad.

2. Ingeniería de Software – Tareas

- **Analizar requisitos y funcionalidad.**
  Interactúa con los clientes para comprender y recopilar requisitos y funcionalidad.
- **Diseñar y desarrollar.**
  escribe código bien estructurado y fácil de mantener que cumpla con los requisitos y se adhiera a los principios de diseño y calidad.
- **Realizar pruebas y depurar.**
  redacta y realiza de pruebas para garantizar que el software sea confiable y libre de errores.
- **Revisar**
  Participa en revisiones de código para mejorar su, garantizar el cumplimiento de los estándares y facilitar el intercambio de conocimiento.
- **Mantener**
  actualiza y mantiene el software, corrige errores y mejora el rendimiento.
- **Documentar**
  redactar documentación, comenta el código, documenta APIs e informes para ayudar a futuros desarrolladores a comprender el sistema.

2. Ingeniería de Software – Factores de calidad

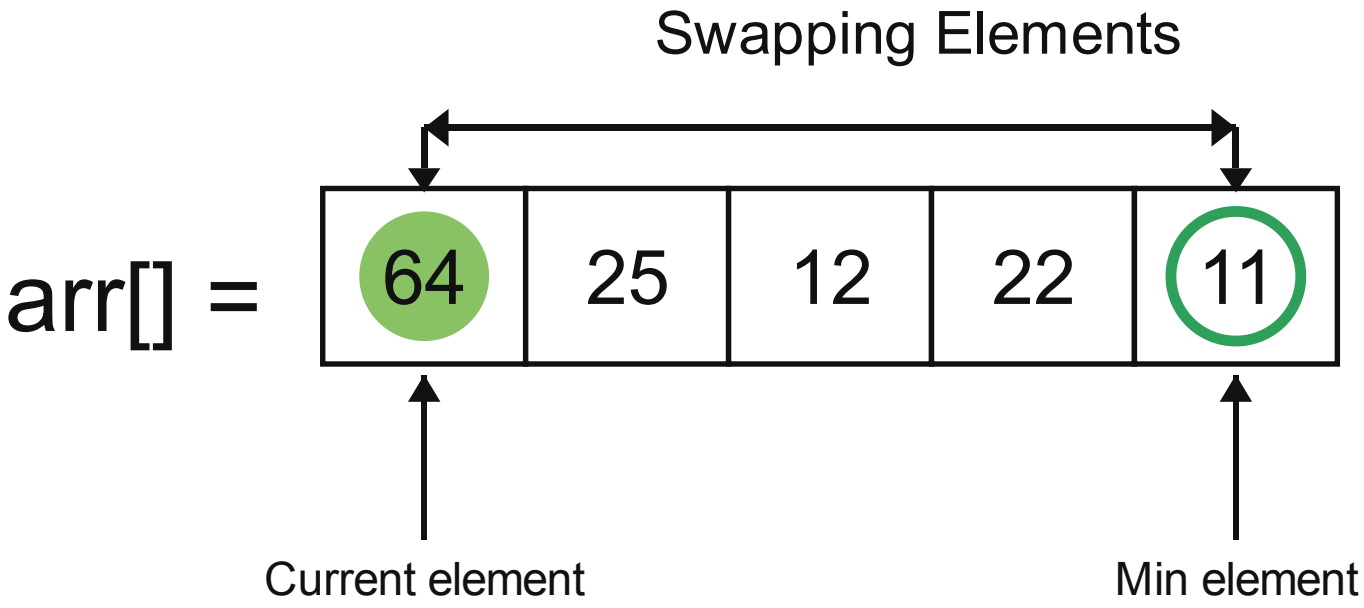
Por **factores de calidad del software** nos referimos a aquellos criterios que podemos utilizar para evaluar de manera separada la calidad general de un producto software y su capacidad para satisfacer las necesidades de los usuarios y adaptarse a diferentes contextos.

Ejemplos:
Corrección, reusabilidad, legibilidad, facilidad de uso, seguridad, etc.

2. Ingeniería de Software – Factores de calidad

Función de ordenación de un vector de enteros.

- Queremos que sea configurable con el orden (ascendente o descendente)
- ¿Cómo lo hacemos?



2. Ingeniería de Software – Factores de calidad

Opción A. Usar un flag.

```
static void sort(int[] list, boolean asc)
{
  ...
  boolean mustSwap;
  if (asc) {
    mustSwap = list[i] > list[j];
  } else {
    mustSwap = list[i] < list[j];
  }
  ...
}
```

Opción B. Usar interfaces

```
@FunctionalInterface
interface Order {
  boolean lessThan(int i, int j);
}

final Order ASCENDING = (i, j) -> i < j;
final Order DESCENDING = (i, j) -> i > j;

static void sort(int[] list, Order order)
{
  ...
  boolean mustSwap = order.lessThan(
    list[j],
    list[i]
  );
  ...
}
```

2. Ingeniería de Software – Calidad externa

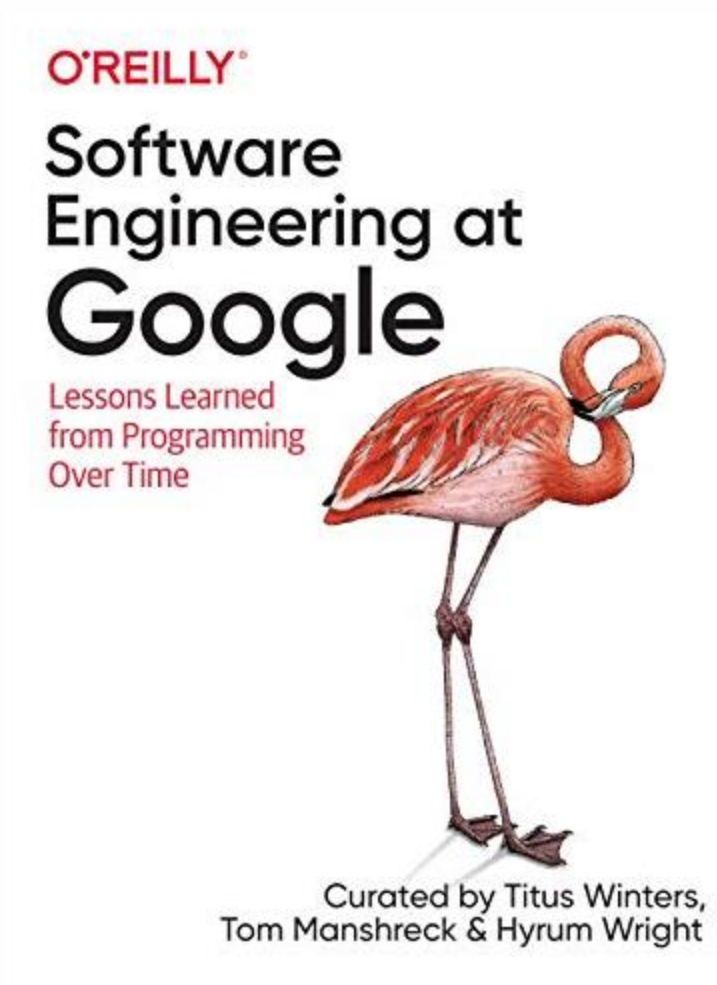
- **Corrección.**
  El software se ajusta a las especificaciones dadas por el usuario.
- **Fiabilidad.**
  Capacidad de ofrecer los mismos resultados bajo las mismas condiciones.
- **No erróneo.**
  No existen diferencias entre los valores reales y los calculados.
- **Eficiencia.**
  Utilización óptima de los recursos de la máquina.
- **Robustez.**
  Comportamiento no catastrófico ante situaciones excepcionales.
- **Portabilidad.**
  Capacidad de integrarse en entornos distintos con un esfuerzo mínimo.

02. Ingeniería de Software – Calidad interna

- **Inteligible.**
  Diseño claro, bien estructurado y documentado.
- **Mantenible.**
  La facilidad de corregir y extender el software.
- **Adaptable (extensible).**
  Modificar alguna función sin que afecte a sus actividades.
- **Reutilizable.**
  El software puede ser usado con facilidad en nuevos desarrollos.

2. Ingeniería de Software – Referencias

- Software Engineering at Google
  - Gratis online: https://abseil.io/resources/swe-book
- Temas del libro:
  - Trabajo en equipo
  - Documentación
  - Revisiones de código
  - Testing
  - Análisis estático
  - Gestión de dependencias
  - Integración continua



03. Procesos de desarrollo

¿Cómo se organiza el desarrollo de software?

3. Procesos – Métodos

Un **método** indica el conjunto de pasos y procedimientos que deben seguirse para el desarrollo de software:

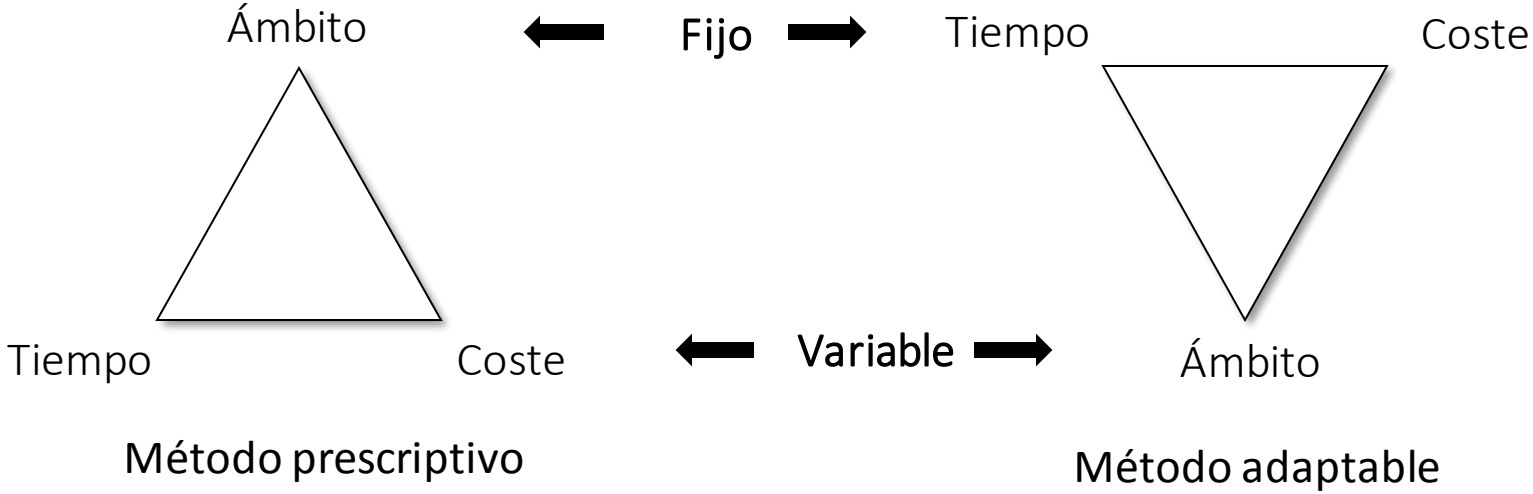
¿Cómo se debe dividir un proyecto en etapas?
¿Qué tareas se llevan a cabo en cada etapa?
Heurísticas para llevar a cabo dichas tareas

¿Qué salidas se producen y cuándo se deben producir ?
¿Qué restricciones se aplican ?
¿Qué herramientas se van a utilizar?
¿Cómo se gestiona y controla un proyecto?

3. Procesos – Métodos

- **Proceso Unificado** (UP, *Unified Process*).
  marco de proceso, orientado a objetos, *heavyweight*, para UML
- **RUP.**
  Rational Unified Process: proceso detallado, instancia UP
- **Larman.**
  simplificado, instancia UP
- **Métrica 3.**
  prescriptivo, para UML y para descomposición funcional (análisis estructurado)
- **Scrum.**
  ágil (*lightweight*, adaptable)
- **Kanban.**
  más flexible todavía *(lean software development)*

3. Procesos – Métodos – Prescriptivos vs adaptables



- **Método prescriptivo.**
  El ámbito es fijo. Con imprevistos, cambia el tiempo y el coste.
- **Método adaptable.**
  El tiempo, coste y la priorización de las características se fija con el cliente.

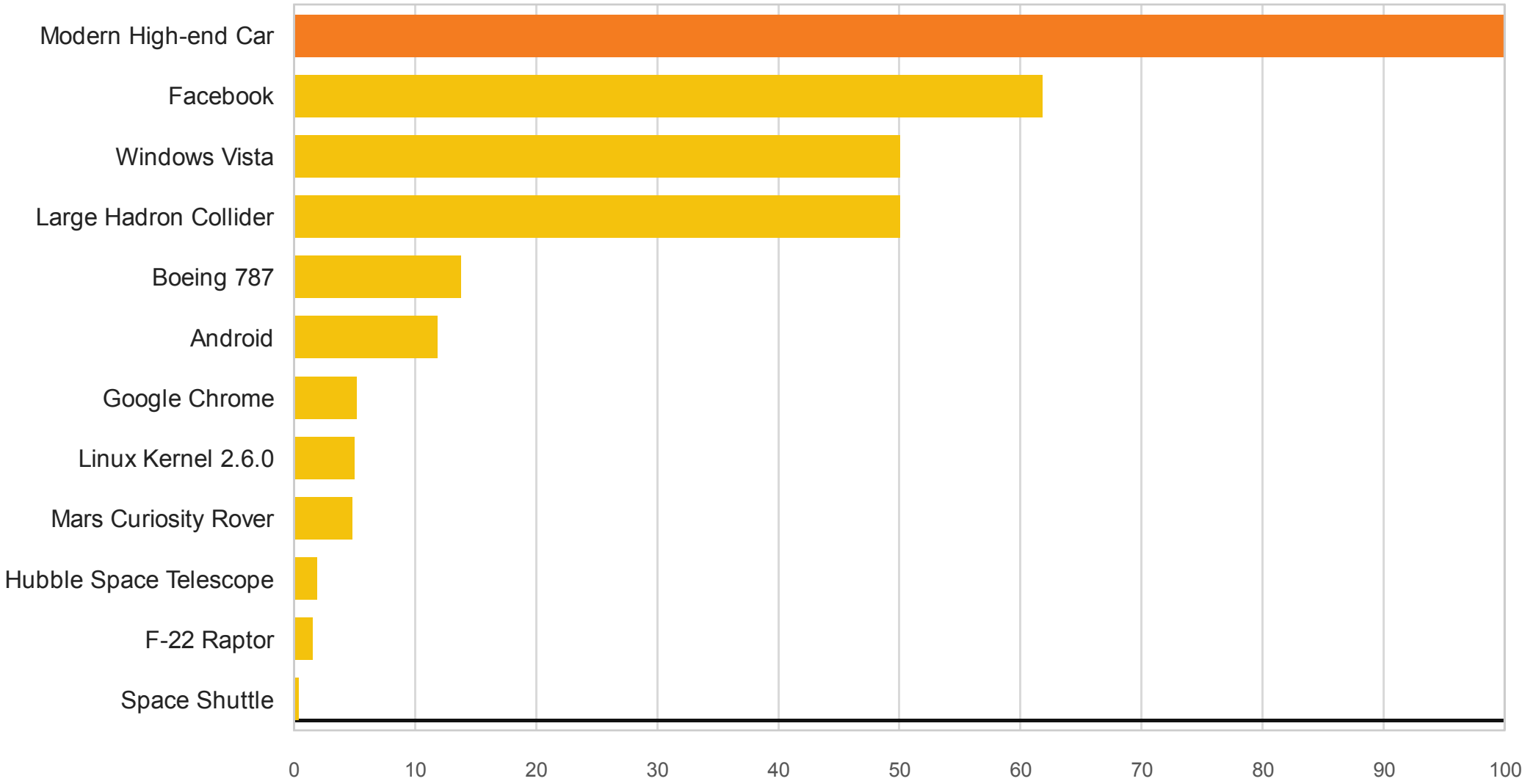
¿Qué ventajas tienen los métodos adaptables?

¿Se utilizan hoy día los métodos prescriptivos? ¿Cuándo tendrían sentido?

4. El software en la actualidad

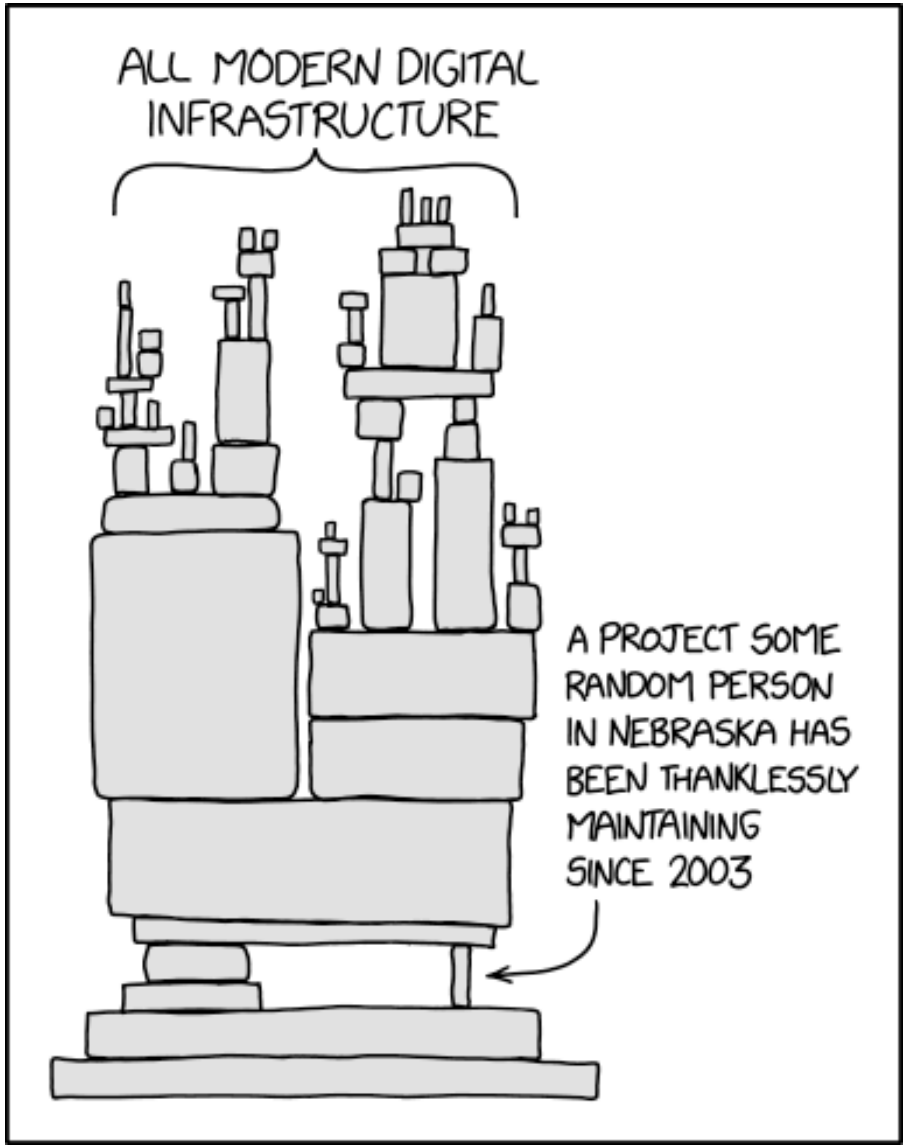
¿En qué han cambiado las cosas en estos últimos años?

4. Software en la actualidad – Estadísticas



4. Software en la actualidad – Motivación

- No es viable escribir todo el código desde cero.
- El software se construye reutilizando componentes.
- La mayor parte del código no la escribe el propio equipo.
- Las librerías dependen de otras librerías. https://npm.anvaka.com/#/view/2d/gifsicle



4. Software en la actualidad – Tendencias

- Modularidad y arquitecturas modernas.
- Reutilización y ecosistema open source.
- Automatización de la verificación y validación.
- Desarrollo en la nube.
- Desarrollo iterativo y gestión del cambio.
- Uso de la inteligencia Artificial.

4. Software en la actualidad – Modularidad

- Énfasis en la modularidad.
  - Permite dividir el trabajo en equipos.
- Arquitecturas y técnicas para conseguir:
  - Alta cohesión, bajo acoplamiento.
  - Escalabilidad.
  - Permitir el cambio.
- APIs para servicios remotos.

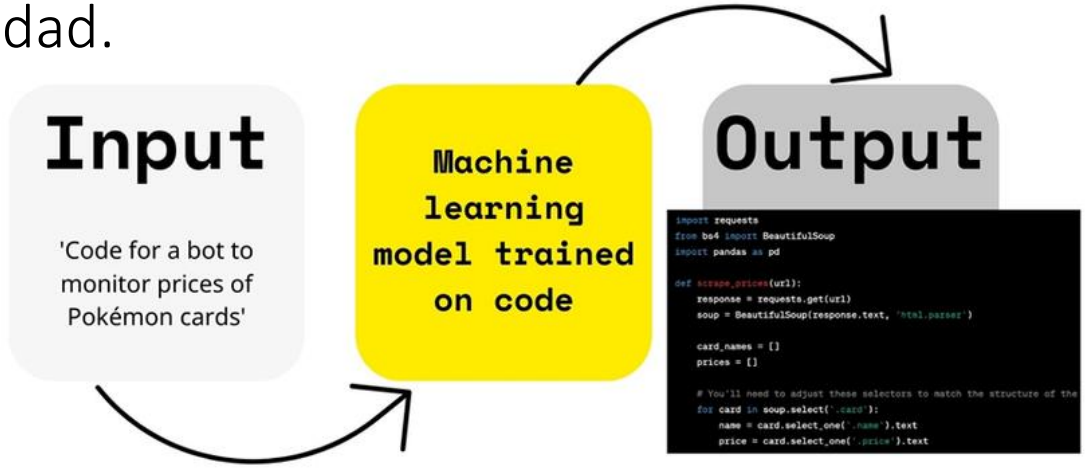
4. Software en la actualidad – Integración continua

- El software se prueba **continuamente**.
  - Probar el software no es negociable
- Se aplican técnicas ligeras de análisis estático
- Herramientas de **Integración Continua (CI)**.
  Cada vez que se sube un cambio al repositorio compartido se ejecutan las actividades de verificación configuradas
  - Ejemplo: GitHub Actions

```
jobs:
  build:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout Code
      uses: actions/checkout@v4

    - name: Set up JDK 17
      uses: actions/setup-java@v4
      with:
        java-version: '17'
        distribution: 'temurin'
        cache: 'maven‘

    - name: Build and Test
      run: mvn --batch-mode test
```

4. Software en la actualidad – Inteligencia Artificial

- La Inteligencia Artificial amplifica la productividad.
  - LLMs son capaces de crear fragmentos de código.
  - Los agentes pueden gestionar proyectos completos.
- La responsabilidad sigue siendo humana.
  Los principios básicos siguen vigentes.
  - Dar instrucciones claras.
  - Revisar el código generado.
  - Verificar calidad, seguridad y mantenibilidad.



Bibliografía

- Ingeniería del software. Un enfoque práctico. 7ª edición. Roger S. Pressman. Mcgraw-Hill.
  - Capítulo 1. El software y la ingeniería de software
  - Capítulo 2. El proceso del software.
- Software Engineering. Basic Principles and Best Practices. Ravi Sethi. Cambridge University Press.
  - Capítulo 2. Software Development Processes