DESARROLLO DE APLICACIONES DE SOFTWARE

HERNÁN ASTUDILLO R.
DEPARTAMENTO DE INFORMÁTICA, UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA

I. SOFTWARE

El término "software" es un anglicismo, inventado en contraposición al "hardware" o ferretería. La industria de TIC[1] mueve alrededor miles de millones de dólares al año en el mundo, y el software está presente en sistemas de todo tipo: de transporte, médicos, de telecomunicaciones, industriales, educacionales, militares, de entretención, oficinas, etc.

En una definición clásica, Pressman (1993) define software como *"... (1) las instrucciones (programas de computadora) que cuando son ejecutadas proveen funciones y el desempeño deseado, (2) las estructuras de datos que permiten a instrucciones manejar adecuadamente la información, y (3) los documentos que describen la operación y uso de las instrucciones."*

Esta definición proporciona algunas pistas sobre las características especiales de la producción de software. Este producto es en esencia información procesando información y entregando como resultado información. Esto lo hace relativamente fácil de replicar y habitualmente muy complejo de idear o desarrollar; en contraste a la parte física de los sistemas de la información (o hardware), donde su complejidad está en la manufactura de los mismos (Carnahan *et al.*, 1997). En consecuencia, dentro de la industria se dice que un software no puede ser fabricado o manufacturado, más bien es desarrollado.

La mayoría de las personas tiene experiencia diaria con software a través de dos canales: su teléfono o "móvil" (si es Android, iPhone o equivalente), y el computador personal "de escritorio" en su oficina u hogar. Si bien tendencias recientes del mercado han tendido a converger ambas categorías (como "tablets" y "laptops"), ambas experiencias aún difieren; en particular, el comportamiento del software móvil en general no es modificable sino a través del cambio de parámetros o adición de "apps", en tanto que el comportamiento del software de escritorio puede modificarse en formas amplias y casi arbitrarias (p.ej. con "macros" de planillas de cálculo o de editores).

[1] TIC: Tecnologías de Información y Comunicación

Sin embargo, desde el punto de vista de la industria, estos dos tipos de software son atípicos: son productos cerrados, vendidos como unidad, y tienen versiones actualizadas cada cierto tiempo, cuya liberación es parte de un ciclo de vida prolongado y guiado por el mercado.

A diferencia de estos tipos de software, los sistemas informáticos de negocios típicamente tienen un "stack" (término inglés para "apilar") con cuatro "capas", cada una de las cuales utiliza a las precedentes:

1) Hardware: máquinas y dispositivos; son típicamente adquiridos *in toto* de grandes proveedores, p.ej. "servidores" (grandes máquinas) HP, IBM o Hitachi.
2) Software básico: disponibiliza las capacidades del hardware independientemente del proveedor de hardware, y es desarrollado por proveedores de gran escala; p.ej. los "sistemas operativos" como Windows (de Microsoft). Linux (de comunidad "open source"), Android (de Google, basado en Linux), y OSX y iOs (ambos de Apple).
3) Middleware: ofrece capacidades especializadas, independientemente del software básico; p.ej. los gestores de bases de datos[2] como DB2 (de IBM), Oracle Database (de Oracle), o MongoDB (de comunidad "open source"); y servidores de aplicaciones, como Oracle WebLogic (de Oracle), WebSphere (de IBM), y JBoss (de RedHat).
4) Aplicaciones: software que ayuda a usuarios concretos a realizar su trabajo; si apoyarán tareas genéricas pueden ser adquiridas (p.ej. Word y Excel, de Microsoft, para edición de documentos y planillas simples), pero si apoyarán tareas únicas o muy complejas deben ser desarrolladas ad-hoc.

Los productos disponibles en estas capas difieren en grado de detalle técnico, en perfil profesional de los especialistas correspondientes, típicamente son adquiridas de proveedores que se mueven en mercados (rubros) diferentes; y especialmente difieren en su grado de estandarización (alto en hardware, muy bajo en aplicaciones).

Las economías de escala han llevado a la adopción de "arquitecturas de referencia" o tecnologías estandarizadas, muchas veces definidas por consorcios de proveedores; así, muchos gestores de bases de datos y servidores de aplicaciones usan la arquitectura JEE (Java Enterprise Edition) y muchas aplicaciones son desarrolladas con esta arquitectura en mente (con el lenguaje de programación Java u otro basado en su JVM); y similarmente para la arquitectura de referencia .NET (controlada por Microsoft). Además, los elementos de software más utilizados típicamente son empacados y vendidos como paquete o "framework".

II. DESARROLLO DE SOFTWARE

El aspecto más riesgos de la construcción de un sistema informático es, siempre, la construcción de elementos ad-hoc, aquello que no puedan ser comprados sino que deban ser

[2] Los gestores de bases de datos (DBMS, por sus siglas en inglés) son casualmente llamados "bases de datos", aunque estrictamente hablando una base de datos es el conjunto de datos almacenados y gestionados por un DBMS.

hechos ad-hos para ese sistema específico. Este aspecto es clave si se desea obtener éxito y rentabilidad repetidamente, y por ello la Ingeniería de Software como disciplina se ocupa de la *construcción sistemática, eficaz y eficiente de sistemas de software*; es decir, de cómo usar recursos limitados para construir software que sea útil a sus clientes y usuarios, y cómo hacer esto una y otra vez. El objeto de estudio es por ende triple: el software (artefactos) y cómo caracterizarlo y mejorarlo; la forma de construirlo (proceso de desarrollo) y cómo caracterizarla y mejorarla; y los grupos estables (organizaciones) que lo construyen eficiente y eficazmente repetidamente, y cómo caracterizarlos y mejorarlos.

Para ilustrar la complejidad y escala del desarrollo de aplicaciones software, utilizaremos una analogía ojalá más cercana a la vida diaria:

- Muchas personas saben cocinar platos básicos (p.ej. arroz).
- Preparar bien un plato elaborado (p.ej. un fricasé) es algo que requiere conocimiento, experiencia y cierta logística.
- Preparar una cena requiere obtener múltiples ingredientes y preparar varios platos simultáneamente, coordinando sus tiempos de término para ajustarse a la secuencia de presentación en la mesa; los platos típicamente pueden ser estipulados informalmente, o incluso consuetudinariamente.
- Preparar un banquete es mucho más complejo: no sólo se debe provisionar y prepar muchas cenas simultáneas, sino que se gestionar un equipo de garzones y espacios ad-hoc; pero además, un banquete debe ser concordado específicamente en platos, restricciones dietéticas, etc. Además de requerir un equipo multi-disciplinario, el factor costo se torna esencial, ya que una cena se puede tratar como un gasto (si es elaborada en casa), pero los banquetes son preparados por organizaciones que deben hacerlos eficientemente una y otra vez.

Las diferencias fundamentales entre ejecutar un plato y ejecutar un banquete son de escala (número de elementos en juego), coordinación de roles (solo v/s un equipo mult-disciplinario), y criterios de éxito (un plato sabroso v/s un evento memorable).

Similarmente en el mundo del software, muchas personas saben programar (p.ej. escribir fórmulas en planillas de cálculo), o incluso escribir programas individuales (p.ej. páginas Web simples), pero no es posible escribir individualmente un **sistema de software**, que consiste en muchos programas que interactúan y colaboran como si fueran una sola gran entidad. P.ej. el "sistema de remuneraciones" de una empresa típicamente tiene programas para registrar horas, sueldos, cargas, adelantos, pagos, imprimir cheques, hacer conciliaciones, ordenar acciones al banco, etc etc.; esta complejidad está oculta a quien sólo recibe un cheque, y es sólo intuída por quienes lo suan para reportar/modificar/imprimir/etc.

La descripción de las tareas y restricciones que un sistema debe proveer y satisfacer (respectivamente) son sus **requisitos** (usual pero erróneamente llamado **"requerimientos"**); las tareas a ejecutar son los **requisitos funcionales** y las restricciones (tiempos mínimos/máximos, cantidades de usuarios, restricciones de seguridad, qué información respaldar, etc.) son los **requisitos extra-funcionales** (a veces mal traducidos con el

anglicismo "requisitos no-funcionales"). En el ejemplo anterior, cada tarea del sistema de remuneraciones debe ser descrita separadamente; el mecanismo usual para describir estas tareas son los "casos de uso", inicialmente introducidos por Jacobson en la década del 60 (a la sazón en la telefónica Ericsson, Suecia) pero progresivamente adoptados por la comunidad de desarrollo de software, y que son hoy en día la práctica estándar (de hecho enseñados a alumnos de Informática en tercer año, sino antes).

III. PROCESOS DE DESARROLLO DE SOFTWARE

La Ingeniería de Software es la disciplina que se ocupa del desarrollo sistemático, eficaz y eficiente de software, y se ocupa de métodos, técnicas, prácticas y estándares tanto para el desarrollo individual como grupal y organizacional. Tal como otras disciplinas tienen su concepto fundamental (así, la célula en biología), el concepto fundamental de la ingeniería de software es una abstracción: el **proceso de desarrollo de software**, que es la descripción ordenada y razonada de la forma en que una persona, grupo u organización aborda sistemáticamente el desarrollo de software.

En los procesos de desarrollo tradicionales, hay tres grandes fases que están presentes en todo proceso de desarrollo de software, estas son: la **definición**, el **desarrollo** y la **mantención**.

1. La **fase de definición** involucra directamente la deducción de **requisitos** del sistema, y permite identificar funcionalidades y caracterizarlas como exigidas, deseadas o preferidas; qué restricciones tendrá el sistema; y qué cualidades o propiedades tendrá el sistema (p.ej. confiabilidad, seguridad, usabilidad, mantenibilidad). Los requisitos deben ser validados por el cliente.

   En esta fase siempre ocurren de alguna forma las siguientes tres actividades:

   1. **Contacto con el cliente**: una actividad de investigación y consulta, para definir qué hará el sistema y qué papel que jugará en el entorno donde funcionará.
   2. **Planificación del proyecto**: consiste en establecer el alcance (funcionalidades, restricciones y propiedades) del sistema, identificar los riesgos, asignar recursos, estimar costos, y definir las tareas y la agenda a seguir.
   3. **Análisis de requisitos**: consiste en establecer una definición más detallada de la información, comportamiento y funciones necesarias en el sistema a construir.

2. La **fase de desarrollo** está enfocada en la construcción del sistema: su arquitectura (componentes y sus relaciones), detalles de código, algoritmos específicos (si los hay), e interfaces humano-computador. En esta fase ocurren las siguientes actividades:

   1. **Diseño:** actividad que convierte los requisitos en **representaciones de software** (también llamada su "diseño") que describen la estructura, arquitectura, algoritmos, e interfaces humano-computador.

2. **Codificación:** Esta etapa se traducen las representaciones del diseño en un lenguaje de programación convencional o no procedimental, que es posible de interpretar por la máquina.
3. **Prueba:** el sistema es "implementado" (instalado en su ambiente objetivo o en un ambiente de prueba similar a él) y ejecutado para identificar errores lógicos, de implementación o de funcionalidad, que son reportados para su corrección.

3. La **fase de mantención** incluye los cambios posteriores a la entrega del sistema, y que a grandes rasgos caen en cuatro categorías:

    1. **De corrección:** algunas funcionalidades o propiedades del sistema pueden estar incorrectamente implementadas (no habiendo sido esto detectado en la etapa de prueba).
    2. **De adaptación:** el entorno en que está instalado el sistemas puede cambiar (p.ej. nuevos computadores), y algunos elementos del sistemas pueden requerir modificaciones.
    3. **De mejora:** algunas funciones adicionales pueden ser detectadas después de comenzar a usar el software (p.ej. agregar una interfaz para móviles).
    4. **De reingeniería:** en ocasiones es necesario reconstruir un software en forma mejorada (p.ej. para simplificar estructura y reducir costos de mantención).

Las tres fases genéricas del desarrollo de software son complementadas con **actividades protectoras**, que se realizan en forma paralela a las fases del proyecto. Algunas actividades protectoras son:

1. **Aseguramiento de la calidad:** considera las actividades ejecutadas en cada actividad, controlando la documentación producida para que esté disponible más tarde.
2. **Gestión de la configuración:** la información de valor creada durante las fases y actividades del proyecto debe ser definida y controlada; diferentes paradigmas de desarrollo proponen diferentes tipos de información a guardar. Existe también un proceso formal de control de cambios para evaluar, aprobar y monitorear cambios a los requisitos durante el desarrollo del proyecto.
3. **Monitoreo del proyecto:** existen varios hitos característicos (fin del análisis, paso a producción, etc.) que permiten observar el progreso de un proyecto; el monitoreo permite verificar si los costos y la agenda están bajo control.
4. **Medición:** tanto el proceso de desarrollo de software ejecutado como el sistema producido pueden ser medidos, con medidas directas (como cantidad de módulos u horas-persona, respectivamente) o indirectas (como puntos de función o nivel de madurez, respectivamente).

En la práctica, la forma específica de ejecutar las fases y actividades varía según los usos y costumbres de cada organización y el paradigma de desarrollo de software que se adopte.

IV. PARADIGMAS DE DESARROLLO DE SOFTWARE

En la historia de la industria y de la disciplina, numerosos procesos han sido propuestos, y aún más han sido usado sin ser jamás descritos formalmente. Para razonar y comparar estos procesos, se reconoce que en general hay unas formas típicas de desarrollar software, llamados **paradigmas** (palabra originalmente introducido porKuhn en 1960 para describir la evolución de la ciencia, pero adoptado por Informática para estos fines). Un **paradigma de desarrollo de software** es un esquema de trabajo para desarrollar producto, que establece procedimientos, hitos y entregables específicos, algunas características clave de calidad del proceso, y el tipo de gestión del proyecto.

La literatura típicamente distingue los paradigmas o modelos de desarrollo de software según el orden de sus actividades. La taxonomía original de Pressman (1993) es aún ampliamente utilizada en la industria (si bien en la academia ha sido desplazado por otras taxonomías razonadas):

1. **Paradigma secuencial — el ciclo de vida clásico**. Basado en el modelo de cascada propuesto por Royce (1970) y modificado para satisfacer algunas restricciones contractuales de adquisiciones (Boehm, 2006). Según la interpretación secuencial — que se ha masificado— inicialmente los requisitos son definidos, después el software es desarrollado (a su vez secuencialmente el diseño, codificación y pruebas), y la mantención sucede posterior a la entrega. Este paradigma es ingenuo y rígido, ya que se ha constatado que los sistemas de grandes en general no pueden ser definidos a priori; en la actualidad, es usado principalmente con fines didácticos o como esqueleto de otros paradigmas.
2. **Prototipado**. Un modelo iterativo de desarrollo de software que privilegia la confianza en la comunicación entre clientes y desarrolladores. Los requisitos son definidos "empíricamente" a través de la creación de modelos (prototipos) que van progresivamente aumentando el nivel de detalle del sistema.
3. **Modelo iterativos (especialmente el "modelo espiral")**. Basado en el ciclo de vida básico y en el desarrollo iterativo o prototipado, plantea una evolución iterativa-incremental del sistema, en que cada nueva iteración está basada en evaluación de valor (por el cliente) y de riesgo (por el desarrollador). Apunta a gestionar proyectos de alto riesgo.

Naturalmente, cada organización adopta un paradigma de desarrollo de software (o combinación de ellos) con fines esencialmente prácticos:

4. Escalonar proyectos como conjuntos regulados de actividades, reduciendo la probabilidad que algunas sean olvidadas o malinterpretadas.
5. Permitir a los clientes comprender y revisar el proyecto, reconocer y validar sus propios roles en él, e identificar áreas de ambigüedad si las hubiere.
6. Los modelos son estratificados, permitiendo el desarrollo progresivo, racional y manejable de los componentes del sistema.

7. Permitir a desarrolladores reconocer sus propios roles en el proyecto, y adoptar o crear notaciones tipográficas adecuadas para representar estructura, funcionalidades, datos y propiedades del sistema.
8. Finalmente, y como beneficio principal, los modelos de proceso permiten estimar la calidad de un sistema en desarrollo antes de esté completo.

Los modelos de proceso estándar son cada vez más detallados, e incluyen criterios que permiten medir la calidad paso a paso, tanto del producto como del proceso, para detección temprana de defectos, como reacción a la antigua práctica de escribir código y revisarlo en acción una vez instalado y (ojalá) funcionando. Por ello, la identificación temprana de los requisitos es esencial para la operación normal de los procesos de desarrollo de software usuales (los procesos "ágiles" prescinden de esto, pero son riesgosos y requieren personal muy motivado y especializado, relegándolos al desarrollo de software de alta exigencia y costo, como *startups* tecnológicas o astroinformática).

Tres tendencias importantes han tomado cuerpo en la última década:

- Boehm (2006), una de las autoridades clave de la disciplina, comenta que desde la década de 1990 la vorágine de la industria ha llevado a privilegiar el *time-to-market* ("tiempo para llegar al mercado"), lo que se refleja en que muchos clientes solicitan productos sin tener claro sus requisitos, dando lugar a **requisitos IKIWISI**: *"I'm not sure, but I'll know it when I see it"* (no estoy seguro, pero sabré cuando lo vea).

- Se han consolidado paradigmas de desarrollo concurrente, con modelos como el espiral o los **métodos ágiles**; los métodos ágiles prescinden de muchas actividades de control y se enfocan en desarrollo iterativo-incremental con iteraciones rápidas y muy ligadas al cliente. Son muy favorecidos por desarrolladores avanzados, pero también muy resistidos por gerencias; en general, en Chile sólo son usados por empresas pequeñas (si bien son los métodos normalmente enseñados en la academia).

- La usabilidad ha cobrado una importancia mucho mayor, ya que los desarrollos son guiados por clientes con productos y nichos específicos en mente; en muchos proyectos de desarrollo de productos para consumidores (como "apps" móviles), la usabilidad es el factor de éxito clave y cuello de botella principal.