Ingeniería del Software I

Requerimientos y Modelado del Software

Ricardo Javier Celi Párraga
Miguel Fabricio Boné Andrade
Aldo Patricio Mora Olivero
Juan Carlos Sarmiento Saavedra

EDITORIAL GRUPO AEA

Ingeniería del Software I

Requerimientos y Modelado del Software

Autores:

Ricardo Javier Celi-Párraga

Miguel Fabricio Boné-Andrade

Aldo Patricio Mora-Olivero

Juan Carlos Sarmiento-Saavedra

Título del libro:
Ingeniería del Software I: Requerimientos y Modelado del Software.
Primera Edición, 2023
Editado en Santo Domingo, Ecuador, 2023
ISBN: 978-9942-7014-7-3

Editado y diseñado por Comité Editorial del Grupo AEA
Hecho e impreso en Santo Domingo - Ecuador

Cita.

Celi Párraga, R. J., Boné Andrade, M. F., Mora Olivero, A. P., Sarmiento Saavedra J. C. (2023). Ingeniería del Software I: Requerimientos y Modelado del Software. Editorial Grupo AEA.

Cada uno de los textos de la Editorial Grupo AEA han sido sometido a un proceso de evaluación por pares doble ciego externos (double-blindpaperreview) con base en la normativa del editorial.

Grupo AEA
Grupo de Asesoría Empresarial y Académica
www.grupo-aea.com
www.editorialgrupo-aea.com
Grupo de Asesoría Empresarial & Académica
Grupoaea.ecuador
Editorial Grupo AEA

Aviso Legal:

La información presentada, así como el contenido, fotografías, gráficos, cuadros, tablas y referencias de este manuscrito es de exclusiva responsabilidad del autor y no necesariamente reflejan el pensamiento de la Editorial Grupo AEA.

Autores

Reseña de Autores

Ricardo Javier Celi Párraga

Universidad Técnica Luis Vargas Torres de Esmeraldas

✉ Correo: ricardo.celi@utelvt.edu.ec

Orcid: https://orcid.org/0000-0002-8525-5744

Ingeniero en sistemas informáticos, Máster en ingeniería del software y sistemas informáticos, Docente de La Universidad Técnica Luis Vargas Torres de Esmeraldas, Sede Santo Domingo de los Tsáchilas.

Miguel Fabricio Boné Andrade

Universidad Técnica Luis Vargas Torres de Esmeraldas

✉Correo: miguel.bone@utelvt.edu.ec

Orcid: https://orcid.org/0000-0002-8635-1869

Ingeniero de sistemas y computación, Magíster en sistemas de telecomunicaciones, Magíster en tecnologías de la información mención en seguridad de redes y comunicaciones, Docente de La Universidad Técnica Luis Vargas Torres de Esmeraldas, Sede Santo Domingo de los Tsáchilas.

Aldo Patricio Mora Olivero

Universidad Técnica Luis Vargas Torres de Esmeraldas

✉Correo: aldo.mora.olivero@utelvt.edu.ec

Orcid: https://orcid.org/0000-0002-4337-7452

Ingeniero en sistemas y computación, Magíster en tecnologías de la información, Docente de La Universidad Técnica Luis Vargas Torres de Esmeraldas, Sede Santo Domingo de los Tsáchilas.

Juan Carlos Sarmiento Saavedra

Universidad Técnica Luis Vargas Torres de Esmeraldas

✉Correo: juan.sarmiento@utelvt.edu.ec

Orcid: http://orcid.org/0000-0001-8114-9410

Ingeniero en sistemas e informática, Magíster en docencia y desarrollo del currículo, Magíster en tecnologías de la información, Docente de La Universidad Técnica Luis Vargas Torres de Esmeraldas, Sede Santo Domingo de los Tsáchilas.

Índice

Introducción

La ingeniería del software es una disciplina que se encarga de aplicar principios de la ingeniería en la creación de software. El objetivo principal de la ingeniería del software es desarrollar software de alta calidad, eficiente y confiable, mediante la aplicación de procesos sistemáticos y metodologías bien definidas.

Para lograr esto, es esencial comprender y satisfacer adecuadamente los requerimientos del software. Los requerimientos son la base de todo proyecto de software, ya que describen las funcionalidades y características que el software debe tener para satisfacer las necesidades de los usuarios y/o clientes.

Para gestionar adecuadamente los requerimientos, es necesario realizar un modelado del software que permita visualizar de manera clara y precisa cómo funcionará el software y cómo interactuará con el usuario. El modelado del software se puede realizar mediante distintos métodos y técnicas, que van desde el uso de diagramas hasta la creación de prototipos y modelos de simulación.

El modelado del software es esencial para el éxito del proyecto, ya que permite detectar posibles errores o deficiencias en la etapa de diseño, evitando costosos errores en la implementación y pruebas. Además, el modelado del software permite comunicar de manera clara y concisa los requerimientos a los distintos miembros del equipo de desarrollo, asegurando una comprensión común y alineada con los objetivos del proyecto.

En resumen, la gestión adecuada de los requerimientos y el modelado del software son elementos esenciales en la ingeniería del software, permitiendo asegurar la calidad, eficiencia y confiabilidad del software desarrollado.

Capítulo I: Introducción a la Ingeniería del Software

Introducción a la Ingeniería del Software

Es imposible operar el mundo moderno sin software. Las infraestructuras nacionales y los servicios públicos se controlan mediante sistemas basados en computadoras, y la mayoría de los productos eléctricos incluyen una computadora y un software de control. La fabricación y la distribución industrial están completamente computarizadas, como el sistema financiero. El entretenimiento, incluida la industria musical, los juegos por computadora, el cine y la televisión, usan software de manera intensiva. Por lo tanto, la ingeniería de software es esencial para el funcionamiento de las sociedades, tanto a nivel nacional como internacional.

Los sistemas de software son abstractos e intangibles. No están restringidos por las propiedades de los materiales, regidos por leyes físicas ni por procesos de fabricación. Esto simplifica la ingeniería de software, pues no existen límites naturales a su potencial. Sin embargo, debido a la falta de restricciones físicas, los sistemas de software pueden volverse rápidamente muy complejos, difíciles de entender y costosos de cambiar.

Hay muchos tipos diferentes de sistemas de software, desde los simples sistemas embebidos, hasta los complejos sistemas de información mundial. No tiene sentido buscar notaciones, métodos o técnicas universales para la ingeniería de software, ya que diferentes tipos de software requieren distintos enfoques. Desarrollar un sistema organizacional de información es completamente diferente de un controlador para un instrumento científico. Ninguno de estos sistemas tiene mucho en común con un juego por computadora de gráficos intensivos. Aunque todas estas aplicaciones necesitan ingeniería de software, no todas requieren las mismas técnicas de ingeniería de software.

1.1. El software

Muchos individuos escriben programas. En las empresas los empleados hacen programas de hoja de cálculo para simplificar su trabajo; científicos e ingenieros elaboran programas para procesar sus datos experimentales, y los aficionados crean programas para su propio interés y satisfacción. Sin embargo, la gran mayoría del desarrollo de software es una actividad profesional, donde el software se realiza para propósitos de negocios específicos, para su inclusión en otros dispositivos o como productos de software, por ejemplo, sistemas de información, sistemas de CAD, etcétera. El software profesional, destinado a usarse por alguien más aparte de su desarrollador, se lleva a cabo en general por equipos, en vez de individualmente. Se mantiene y cambia a lo largo de su vida.

La ingeniería de software busca apoyar el desarrollo de software profesional, en lugar de la programación individual. Incluye técnicas que apoyan la especificación, el diseño y la evolución del programa, ninguno de los cuales son normalmente relevantes para el desarrollo de software personal.

Muchos suponen que el software es tan sólo otra palabra para los programas de cómputo. No obstante, cuando se habla de ingeniería de software, esto no sólo se refiere a los programas en sí, sino también a toda la documentación asociada y los datos de configuración requeridos para hacer que estos programas operen de manera correcta. Un sistema de software desarrollado profesionalmente es usualmente más que un solo programa.

Ésta es una de las principales diferencias entre el desarrollo de software profesional y el de aficionado. Si usted diseña un programa personal, nadie más lo usará ni tendrá que preocuparse por elaborar guías del programa, documentar el diseño del programa, etcétera. Por el contrario, si crea software que otros usarán y otros ingenieros cambiarán, entonces, en general debe ofrecer información adicional, así como el código del programa.

1.2. Evolución del software

El contexto en que se ha desarrollado el software está fuertemente ligado a las casi cinco décadas de evolución de los sistemas informáticos. Un mejor rendimiento del hardware, una reducción del tamaño y un coste más bajo, han dado lugar a sistemas informáticos más sofisticados. A continuación, se describe la evolución del Software dentro del contexto de las áreas de aplicación de los sistemas basados en computadoras.

1) **Los primeros años (1950 - 1965):**
   - El software estaba en su infancia
   - El software era un añadido
   - Existían pocos métodos para la programación
   - No se tenía una planificación para el desarrollo del software
   - Los programadores trataban de hacer las cosas bien
   - El software se diseñaba a medida
   - El software era desarrollado y utilizado por la misma persona u organización (entorno personalizado)
   - El diseño de software era realizado en la mente de alguien y no existía documentación
2) **La segunda era (1965 - 1975):**
   - Multiprogramación y sistemas multiusuarios introducen nuevos conceptos de interacción hombre-máquina.
   - Sistemas de tiempo real que podían recoger, analizar y transformar datos de múltiples fuentes.
   - Avances en los dispositivos de almacenamiento en línea condujeron a la primera generación de sistemas de gestión de Base de Datos.
   - Software como producto y la llegada de las "casas de software" produciéndose así una amplia distribución en el mercado.
   - El software se desarrollaba para ser comercializado
   - Se empezó a distribuir software para grandes computadoras y minicomputadores
   - El mantenimiento de software comenzó a absorber recursos en una gran medida.

- Comenzó una crisis del software porque la naturaleza personalizada de los programas hizo imposible su mantenimiento.
- Conforme crecía el número de sistemas informáticos, comenzaron a extenderse las bibliotecas de software de computadora. Las casas desarrollaban proyectos en que se producían programas de decenas de miles de sentencias fuente. Los productos de software comprados en el exterior incorporaban cientos de miles de nuevas sentencias. Una nube negra apareció en el horizonte. Todos estos programas tenían que ser corregidos cuando se detectaban fallos, modificados cuando cambiaban los requisitos de los usuarios o adaptados a nuevos dispositivos de hardware que se hubiera adquirido. Estas actividades se llamaron colectivamente mantenimiento del software.

3) La tercera era (1975 - 1985):

- Procesamiento Distribuido. Múltiples computadoras, cada una ejecutando funciones concurrentes y comunicándose con alguna otra.
- Redes de área local y de área global. Comunicaciones digitales de alto ancho de banda y la creciente demanda de acceso "instantáneo" a los datos.
- Amplio uso de microprocesadores y computadoras personales (hardware de bajo costo). Incorporación de "inteligencia" (autos, hornos de microondas, robots industriales y equipos de diagnóstico de suero sanguíneo). Impacto en el consumo.
- Planificación en el proceso del desarrollo de software.

4) La cuarta era (1985 -2000):

- Tecnología orientada a objetos
- Los sistemas expertos y la inteligencia artificial se han trasladado del laboratorio a las aplicaciones prácticas.
- Software para redes neuronales artificiales (simulación de procesamiento de información al estilo de como lo hacen los humanos).
- Impacto colectivo del software
- Sistemas operativos sofisticados, en redes globales y locales
- Aplicaciones de software avanzadas

- Entorno cliente/cliente servidor
- Superautopista de información y una conexión del ciberespacio
- La industria del software es la cuna de la economía
- Técnicas de cuarta generación para el desarrollo de software
- Programación de realidad virtual y sistemas multimedia
- Algoritmos genéticos
- Adopción de prácticas de Ingeniería del software

5) **La quinta era (2000-hoy):**
   - Utiliza algunos requisitos de las eras anteriores.
   - Aumenta la presencia de la web
   - Software para teléfonos móviles
   - Reutilización de información
   - Reutilización de componentes

1.3. Crisis del software

El término “Crisis del Software” fue acuñado a principios de los años 70, cuando la ingeniería de software era prácticamente inexistente. El término expresaba las dificultades del desarrollo de software frente al rápido crecimiento de la demanda por software, de la complexidad de los problemas a ser resueltos y de la inexistencia de técnicas establecidas para el desarrollo de sistemas que funcionaran adecuadamente o pudieran ser validados.

Dificultad en escribir programas libres de defectos, fácilmente comprensibles, y que sean verificables (Dijkstra–1968)

Causas:

- Los proyectos no terminaban en plazo
- Los proyectos no se ajustaban al presupuesto inicial
- Software que no cumplía las especificaciones
- Código inmantenible que dificultaba la gestión y evolución del proyecto

Consecuencias:

- Baja Calidad del Software

- Tiempo y Presupuesto Excedido
- Confiabilidad Cuestionable
- Altos requerimientos del personal para el desarrollo y el mantenimiento

1.4. Mitos del software

Existieron muchas causas de la crisis del software, esto se pueden encontrar en una mitología que surge durante los primeros años del desarrollo del software. Hoy, la mayoría de los profesionales competentes consideran a los mitos como actitudes erróneas que han causado serios problemas, tanto a los gestores como a los técnicos. Sin embargo, las viejas actitudes y hábitos son difíciles de modificar, y todavía se cree en algunos restos de los mitos del software.

1.4.1. Mitos de gestión.

Los gestores con responsabilidad sobre el software, como los gestores en la mayoría de las disciplinas, están normalmente bajo la presión de cumplir los presupuestos, hacer que no se retrase el proyecto y mejorar la calidad.

- **Mito:** Tenemos ya un libro que está lleno de estándares y procedimientos para construir software. ¿No le proporciona ya a mi gente todo lo que necesita saber?

  Realidad: Esta muy bien que el libro exista, pero ¿se usa?, ¿conocen los trabajadores su existencia?, ¿refleja las prácticas modernas de desarrollo de software?, ¿es completo? En muchos casos, la respuesta a todas estas preguntas es "no".

- **Mito**: Mi gente dispone de las herramientas de desarrollo de software más avanzadas, después de todo, les compramos las computadoras más modernas.

  Realidad: Se necesita mucho más que el último modelo de computadora grande (o de PC) para hacer desarrollo de software de gran calidad. Las herramientas de ingeniería del software asistida por computadora (CASE), aunque la mayoría todavía no se usen, son más importantes que el hardware para conseguir buena calidad y productividad.

- **Mito**: Si fallamos en la planificación, podemos añadir más programadores y adelantar el tiempo perdido.

  Realidad: El desarrollo de software no es un proceso mecánico como la fabricación. En otras palabras, añadir gente a un proyecto de software retrasado lo retrasa aún más>>. Al principio, esta declaración puede parecer un contra sentido. Sin embargo, cuando se añaden nuevas personas, le necesidad de aprender y comunicarse con el equipo puede y hace que se reduzca la cantidad de tiempo gastado en el desarrollo productivo. Puede añadirse gente, pero sólo de una manera planificada y bien coordinada.

1.4.2. Mitos del cliente.

Un cliente que solicita una aplicación de software puede ser una persona del despacho de al lado, un grupo técnico de la sala de abajo, el departamento de ventas o una compañía exterior que solicita un software bajo contrato. Los mitos conducen a que el cliente se cree una falsa expectativa y finalmente, quede insatisfecho con el que desarrolla el software.

- **Mito**: Una declaración general de los objetivos es suficiente para comenzar a escribir los programes; podemos dar los detalles más adelante.

  Realidad: Una mala definición inicial es la principal causa del trabajo baldío en software. Es esencial una descripción formal y detallada del ámbito de la información, funciones, rendimiento, interfaces, ligaduras del diseño y criterios de validación. Estas características pueden determinarse sólo después de una exhaustiva comunicación entre el cliente y el analista.

- **Mito**: Los requisitos del proyecto cambian continuamente, pero los cambios pueden acomodarse fácilmente, ya que el software es flexible.

  Realidad: Es verdad que los requisitos del software cambian, pero el impacto del cambio varía según el momento en que se introduzca. Si se pone cuidado al dar la definición inicial, los cambios solicitados al

principio pueden acomodarse fácilmente. El cliente puede revisar los requisitos y recomendar las modificaciones con relativamente poco impacto en el costo. Cuando los cambios se solicitan durante el diseño del software, el impacto en el costo crece rápidamente. Ya se han acordado los recursos a utilizar y se ha establecido un esqueleto del diseño. Los cambios pueden producir trastornos que requieran recursos adicionales e importantes modificaciones del diseño; es decir, costo adicional. Los cambios en la función, rendimiento, interfaces u otras características, impacto importante sobre el costo. Cuando se solicitan al final de un proyecto, los cambios pueden producir un orden de magnitud más caro que el mismo cambio pedido al principio.

1.4.3. Mitos de los desarrolladores.

Los mitos en los que aún creen muchos desarrolladores se han ido fomentando durante cuatro décadas de cultura informática. Durante los primeros días del desarrollo del software, la programación se veía como un arte. Las viejas formas y actitudes tardan en morir.

- **Mito**: Una vez que escribimos el programa y hacemos que funcione, nuestro trabajo ha terminado.

  Realidad: Los datos industriales indican que entre el 50 % y el 60% de todo el esfuerzo dedicado a un programa se realizará después de que se le haya entregado al cliente por primera vez.

- **Mito**: Hasta que no tenga el programa << ejecutándose >> realmente no tengo forma de comprobar su calidad.

  Realidad: Desde el principio del proyecto se puede aplicar uno de los mecanismos más efectivos para garantizar la calidad del software: la revisión técnica formal. La revisión del software es un << filtro de calidad >> que se ha comprobado que es más efectivo que la prueba, para encontrar ciertas clases de defectos en el software.

- **Mito**: Lo único que se entrega al terminar el proyecto es el programa funcionando.

Realidad: Un programa funcionando es sólo parte de una configuración del software que incluye programas, documentos, y datos. La documentación es la base de un buen desarrollo y, lo que es más importante, proporciona guías para la tarea de mantenimiento del software.

1.5. Características del software

Mantenimiento

El software debe escribirse de tal forma que pueda evolucionar para satisfacer las necesidades cambiantes de los clientes. Éste es un atributo crítico porque el cambio del software es un requerimiento inevitable de un entorno empresarial variable.

Confiabilidad y seguridad

La confiabilidad del software incluye un rango de características que abarcan fiabilidad, seguridad y protección. El software confiable no tiene que causar daño físico ni económico, en caso de falla del sistema. Los usuarios malintencionados no deben tener posibilidad de acceder al sistema o dañarlo.

Eficiencia

El software no tiene que desperdiciar los recursos del sistema, como la memoria y los ciclos del procesador. Por lo tanto, la eficiencia incluye capacidad de respuesta, tiempo de procesamiento, utilización de memoria, etcétera.

Aceptabilidad

El software debe ser aceptable al tipo de usuarios para quienes se diseña. Esto significa que necesita ser comprensible, utilizable y compatible con otros sistemas que ellos usan.

1.6. Tipos de software

Los ingenieros de software están interesados por el desarrollo de productos de software (es decir, software que puede venderse a un cliente). Existen dos tipos de productos de software:

Productos genéricos

Consisten en sistemas independientes que se producen por una organización de desarrollo y se venden en el mercado abierto a cualquier cliente que desee comprarlos. Ejemplos de este tipo de productos incluyen software para PC, como bases de datos, procesadores de texto, paquetes de dibujo y herramientas de administración de proyectos. También abarcan las llamadas aplicaciones verticales diseñadas para cierto propósito específico, tales como sistemas de información de librería, sistemas de contabilidad o sistemas para mantener registros dentales.

Productos personalizados (o a la medida)

Son sistemas que están destinados para un cliente en particular. Un contratista de software desarrolla el programa especialmente para dicho cliente. Ejemplos de este tipo de software incluyen los sistemas de control para dispositivos electrónicos, sistemas escritos para apoyar cierto proceso empresarial y los sistemas de control de tráfico aéreo.

1.7. Dominios de aplicación del software

Software de sistemas: conjunto de programas escritos para dar servicio a otros programas. Determinado software de sistemas (por ejemplo, compiladores, editores y herramientas para administrar archivos) procesa estructuras de información complejas pero deterministas. Otras aplicaciones de sistemas (por ejemplo, componentes de sistemas operativos, manejadores, software de redes, procesadores de telecomunicaciones) procesan sobre todo datos indeterminados. En cualquier caso, el área de software de sistemas se caracteriza por: gran interacción con el hardware de la computadora, uso intensivo por parte de usuarios múltiples, operación concurrente que requiere la

secuenciación, recursos compartidos y administración de un proceso sofisticado, estructuras complejas de datos e interfaces externas múltiples.

Software de aplicación: programas aislados que resuelven una necesidad específica de negocios. Las aplicaciones en esta área procesan datos comerciales o técnicos en una forma que facilita las operaciones de negocios o la toma de decisiones administrativas o técnicas. Además de las aplicaciones convencionales de procesamiento de datos, el software de aplicación se usa para controlar funciones de negocios en tiempo real (por ejemplo, procesamiento de transacciones en punto de venta, control de procesos de manufactura en tiempo real).

Software de ingeniería y ciencias: se ha caracterizado por algoritmos "devoradores de números". Las aplicaciones van de la astronomía a la vulcanología, del análisis de tensiones en automóviles a la dinámica orbital del transbordador espacial, y de la biología molecular a la manufactura automatizada. Sin embargo, las aplicaciones modernas dentro del área de la ingeniería y las ciencias están abandonando los algoritmos numéricos convencionales. El diseño asistido por computadora, la simulación de sistemas y otras aplicaciones interactivas, han comenzado a hacerse en tiempo real e incluso han tomado características del software de sistemas.

Software incrustado: reside dentro de un producto o sistema y se usa para implementar y controlar características y funciones para el usuario final y para el sistema en sí. El software incrustado ejecuta funciones limitadas y particulares (por ejemplo, control del tablero de un horno de microondas) o provee una capacidad significativa de funcionamiento y control (funciones digitales en un automóvil, como el control del combustible, del tablero de control y de los sistemas de frenado).

Software de línea de productos: es diseñado para proporcionar una capacidad específica para uso de muchos consumidores diferentes. El software de línea de productos se centra en algún mercado limitado y particular (por ejemplo, control del inventario de productos) o se dirige a mercados masivos de consumidores (procesamiento de textos, hojas de cálculo, gráficas por computadora,

multimedios, entretenimiento, administración de base de datos y aplicaciones para finanzas personales o de negocios).

Aplicaciones web: llamadas “webapps”, esta categoría de software centrado en redes agrupa una amplia gama de aplicaciones. En su forma más sencilla, las webapps son poco más que un conjunto de archivos de hipertexto vinculados que presentan información con uso de texto y gráficas limitadas. Sin embargo, desde que surgió Web 2.0, las webapps están evolucionando hacia ambientes de cómputo sofisticados que no sólo proveen características aisladas, funciones de cómputo y contenido para el usuario final, sino que también están integradas con bases de datos corporativas y aplicaciones de negocios.

Software de inteligencia artificial: hace uso de algoritmos no numéricos para resolver problemas complejos que no son fáciles de tratar computacionalmente o con el análisis directo. Las aplicaciones en esta área incluyen robótica, sistemas expertos, reconocimiento de patrones (imagen y voz), redes neurales artificiales, demostración de teoremas y juegos.

1.8. Ingeniería del software

La ingeniería de software es una disciplina de ingeniería que se interesa por todos los aspectos de la producción de software, desde las primeras etapas de la especificación del sistema hasta el mantenimiento del sistema después de que se pone en operación. En esta definición se presentan dos frases clave:

Disciplina de ingeniería

Los ingenieros hacen que las cosas funcionen. Aplican teorías, métodos y herramientas donde es adecuado. Sin embargo, los usan de manera selectiva y siempre tratan de encontrar soluciones a problemas, incluso cuando no hay teorías ni métodos aplicables. Los ingenieros también reconocen que deben trabajar ante restricciones organizacionales y financieras, de modo que buscan soluciones dentro de tales limitaciones.

Todos los aspectos de la producción del software

La ingeniería de software no sólo se interesa por los procesos técnicos del desarrollo de software, sino también incluye actividades como la administración del proyecto de software y el desarrollo de herramientas, así como métodos y teorías para apoyar la producción de software.

La ingeniería busca obtener resultados de la calidad requerida dentro de la fecha y del presupuesto. A menudo esto requiere contraer compromisos: los ingenieros no deben ser perfeccionistas. Sin embargo, las personas que diseñan programas para sí mismas podrían pasar tanto tiempo como deseen en el desarrollo del programa. En general, los ingenieros de software adoptan en su trabajo un enfoque sistemático y organizado, pues usualmente ésta es la forma más efectiva de producir software de alta calidad. No obstante, la ingeniería busca seleccionar el método más adecuado para un conjunto de circunstancias y, de esta manera, un acercamiento al desarrollo más creativo y menos formal sería efectivo en ciertas situaciones.

1.9. Ética en la ingeniería de software

Como otras disciplinas de ingeniería, la ingeniería de software se realiza dentro de un marco social y legal que limita la libertad de la gente que trabaja en dicha área. Como ingeniero de software, usted debe aceptar que su labor implica responsabilidades mayores que la simple aplicación de habilidades técnicas. También debe comportarse de forma ética y moralmente responsable para ser respetado como un ingeniero profesional.

No sobra decir que debe mantener estándares normales de honestidad e integridad. No debe usar sus habilidades y experiencia para comportarse de forma deshonesta o de un modo que desacredite la profesión de ingeniería de software. Sin embargo, existen áreas donde los estándares de comportamiento aceptable no están acotados por la legislación, sino por la noción más difusa de responsabilidad profesional. Algunas de ellas son:

1. **Confidencialidad.** Por lo general, debe respetar la confidencialidad de sus empleadores o clientes sin importar si se firmó o no un acuerdo formal sobre la misma.
2. **Competencia.** No debe desvirtuar su nivel de competencia. Es decir, no hay que aceptar de manera intencional trabajo que esté fuera de su competencia.
3. **Derechos de propiedad intelectual.** Tiene que conocer las leyes locales que rigen el uso de la propiedad intelectual, como las patentes y el copyright. Debe ser cuidadoso para garantizar que se protege la propiedad intelectual de empleadores y clientes.
4. **Mal uso de computadoras.** No debe emplear sus habilidades técnicas para usar incorrectamente las computadoras de otros individuos. El mal uso de computadoras varía desde lo relativamente trivial (esto es, distraerse con los juegos de la PC del compañero) hasta lo extremadamente serio (diseminación de virus u otro malware).

1.10. Proyectos de software

Un proyecto software es todo el procedimiento del desarrollo de software, desde la recogida de requisitos, pasando por las pruebas y el mantenimiento, y llevado a cabo en acorde a las metodologías de ejecución, en un momento concreto en el tiempo para lograr el producto software deseado.

Proyecto Exitoso

Factores que hay que tener en cuenta para que un proyecto sea exitoso

- **Ejecución del proyecto**
  - Terminado a tiempo
  - Ajustado al presupuesto
  - Documentado
  - No deben haber personas claves
  - Participación activa del usuario y de todos los involucrados
  - Metodología
  - Experiencia
  - Puesta en marcha exitosa

Factores que hay que tener en cuenta para que un producto de software sea exitoso.

- **Producto**
  - El bien o servicio es coherente con su fin (objetivos logrados)
  - El producto cumple las funcionalidades esperadas
  - Es flexible (modificable)
  - Es confiable
  - Es fácil de usar

Proyecto Fracasado

A continuación, se mencionan algunas malas prácticas que hacen que un proyecto de software fracase.

- **Ejecución del proyecto**
  - No terminado en el plazo señalado
  - Falta de participación (los usuarios no se involucraron)
  - No documentado (por falta de tiempo)
  - Personal clave desaparece

- **Producto**
  - 90% de la funcionalidad terminada (Ley de pareto).
  - No existen manuales o la documentación del sistema asociada

Evolución de un proyecto fracasado

Factores del fracaso de proyectos

- Requerimientos vagos o no definidos
- Mala planificación
- Supuestos mal definidos
- Recursos irreales (físicos, humanos)
- Continuo cambio de requerimientos (cambios día a día)
- Mal manejo o administración de cambios
- Malos recursos
- Mala participación
- Falta de lenguaje común
- Falta de un líder

Conclusiones

- No todos los proyectos son exitosos.
- El éxito de un proyecto depende en gran porcentaje de la dirección
- El dirigir un proyecto es atacar las fuentes de problemas y fomentar los factores de éxito.
- La dirección de un proyecto y la ejecución parecen iguales, pero son de naturaleza distinta.

Capítulo II: El Proceso del Software

El Proceso del Software

2.1. Modelos de proceso de software

Un proceso de software es una serie de actividades relacionadas que conduce a la elaboración de un producto de software. Estas actividades pueden incluir el desarrollo de software desde cero en un lenguaje de programación estándar como Java o C. Sin embargo, las aplicaciones de negocios no se desarrollan precisamente de esta forma. El nuevo software empresarial con frecuencia ahora se desarrolla extendiendo y modificando los sistemas existentes, o configurando e integrando el software comercial o componentes del sistema.

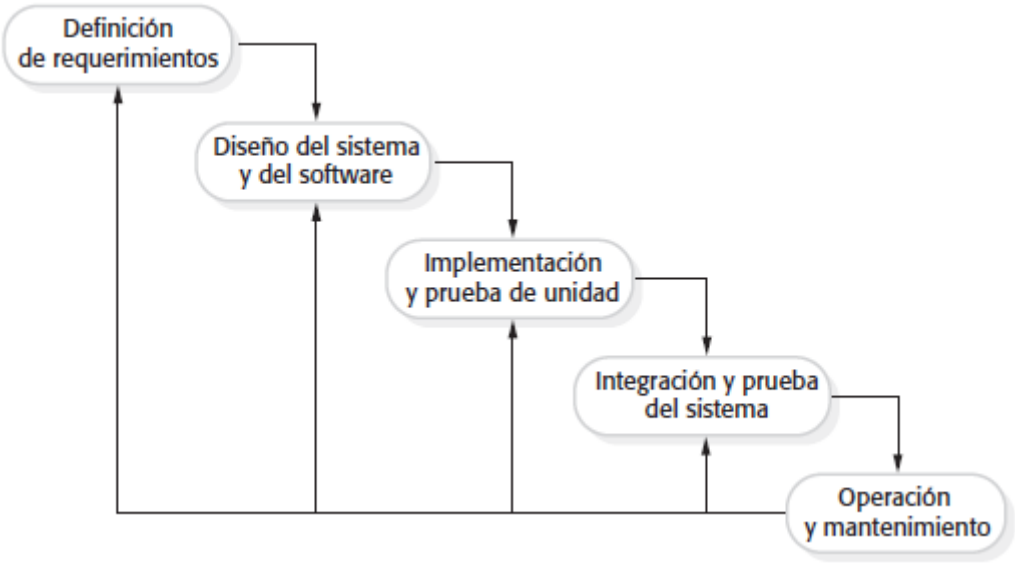
Existen muchos diferentes procesos de software. Según Somerville (2011) todos los procesos deben incluir cuatro actividades que son fundamentales para la ingeniería de software.

Ciclo de vida del software:

1. **Especificación del software** Tienen que definirse tanto la funcionalidad del software como las restricciones de su operación.
2. **Diseño e implementación del software** Debe desarrollarse el software para cumplir con las especificaciones.
3. **Validación del software** Hay que validar el software para asegurarse de que cumple lo que el cliente quiere.
4. **Evolución del software** El software tiene que evolucionar para satisfacer las necesidades cambiantes del cliente.

2.1.1. El modelo en cascada

Éste toma las actividades fundamentales del proceso de especificación, desarrollo, validación y evolución y, luego, los representa como fases separadas del proceso, tal como especificación de requerimientos, diseño de software, implementación, pruebas, etcétera.



Las principales etapas del modelo en cascada reflejan directamente las actividades fundamentales del desarrollo:

1. Análisis y definición de requerimientos Los servicios, las restricciones y las metas del sistema se establecen mediante consulta a los usuarios del sistema. Luego, se definen con detalle y sirven como una especificación del sistema.

2. Diseño del sistema y del software El proceso de diseño de sistemas asigna los requerimientos, para sistemas de hardware o de software, al establecer una arquitectura de sistema global. El diseño del software implica identificar y describir las abstracciones fundamentales del sistema de software y sus relaciones.

3. Implementación y prueba de unidad Durante esta etapa, el diseño de software se realiza como un conjunto de programas o unidades del programa. La prueba de unidad consiste en verificar que cada unidad cumpla con su especificación.

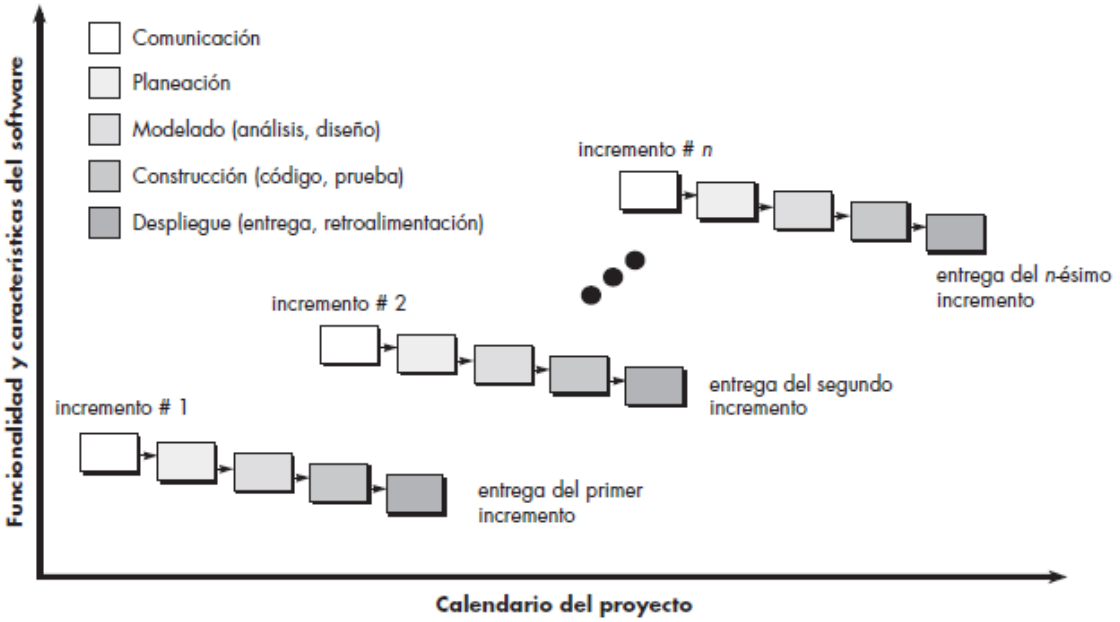
4. Integración y prueba de sistema Las unidades del programa o los programas individuales se integran y prueban como un sistema completo para asegurarse de que se cumplan los requerimientos de software. Después de probarlo, se libera el sistema de software al cliente.

5. Operación y mantenimiento Por lo general (aunque no necesariamente), ésta es la fase más larga del ciclo de vida, donde el sistema se instala y se pone en práctica. El mantenimiento incluye corregir los errores que no se detectaron en etapas anteriores del ciclo de vida, mejorar la implementación de las unidades del sistema e incrementar los servicios del sistema conforme se descubren nuevos requerimientos.

2.1.2. Modelo incremental

El modelo de proceso incremental se centra en que en cada incremento se entrega un producto que ya opera. Los primeros incrementos son versiones desnudas del producto final, pero proporcionan capacidad que sirve al usuario y también le dan una plataforma de evaluación.

El desarrollo incremental es útil en particular cuando no se dispone de personal para la implementación completa del proyecto en el plazo establecido por el negocio. Los primeros incrementos se desarrollan con pocos trabajadores. Si el producto básico es bien recibido, entonces se agrega más personal (si se requiere) para que labore en el siguiente incremento. Además, los incrementos se planean para administrar riesgos técnicos. Por ejemplo, un sistema grande talvez requiera que se disponga de hardware nuevo que se encuentre en desarrollo y cuya fecha de entrega sea incierta. En este caso, tal vez sea posible planear los primeros incrementos de forma que eviten el uso de dicho hardware, y así proporcionar una funcionalidad parcial a los usuarios finales sin un retraso importante.



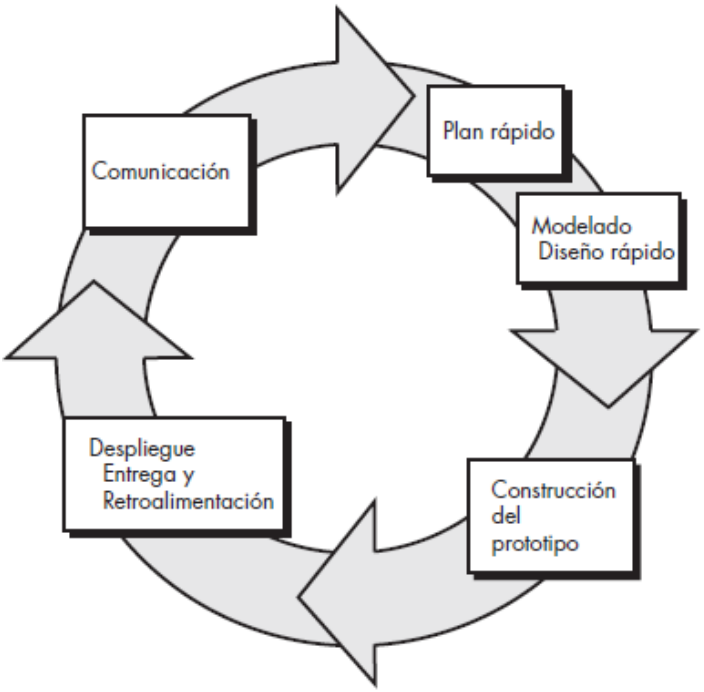
2.1.3. Modelo evolutivo (prototipos)

Los modelos evolutivos son iterativos. Se caracterizan por la manera en la que permiten desarrollar versiones cada vez más completas del software. En los párrafos que siguen se presentan dos modelos comunes de proceso evolutivo.

Hacer prototipos. Es frecuente que un cliente defina un conjunto de objetivos generales para el software, pero que no identifique los requerimientos detallados para las funciones y características. En otros casos, el desarrollador tal vez no esté seguro de la eficiencia de un algoritmo, de la adaptabilidad de un sistema operativo o de la forma que debe adoptar la interacción entre el humano y la máquina. En estas situaciones, y muchas otras, el paradigma de hacer prototipos tal vez ofrezca el mejor enfoque.

El paradigma de hacer prototipos comienza con comunicación. Usted se reúne con otros participantes para definir los objetivos generales del software, identifica cualesquiera requerimientos que conozca y detecta las áreas en las que es imprescindible una mayor definición. Se planea rápidamente una iteración para hacer el prototipo, y se lleva a cabo el modelado (en forma de un “diseño rápido”). Éste se centra en la representación de aquellos aspectos del software que serán visibles para los usuarios finales (por ejemplo, disposición de la interfaz humana

o formatos de la pantalla de salida). El diseño rápido lleva a la construcción de un prototipo. Éste se entrega y es evaluado por los participantes, que dan retroalimentación para mejorar los requerimientos. La iteración ocurre a medida que el prototipo es afinado para satisfacer las necesidades de distintos participantes, y al mismo tiempo le permite a usted entender mejor lo que se necesita hacer (Pressman, 2010).



2.1.4. El modelo espiral

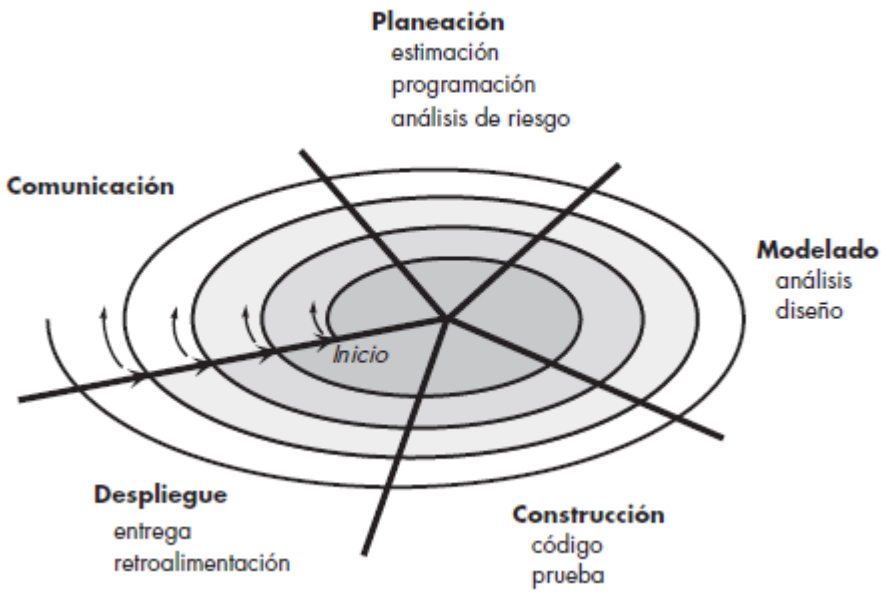
El modelo espiral. Propuesto en primer lugar por Barry Boehm, el modelo espiral es un modelo evolutivo del proceso del software y se acopla con la naturaleza iterativa de hacer prototipos con los aspectos controlados y sistémicos del modelo de cascada. Tiene el potencial

para hacer un desarrollo rápido de versiones cada vez más completas. Boehm describe el modelo del modo siguiente:

> El modelo de desarrollo espiral es un generador de modelo de proceso impulsado por el riesgo, que se usa para guiar la ingeniería concurrente con participantes múltiples de sistemas intensivos en software. Tiene dos características distintivas principales. La primera es el enfoque cíclico para el crecimiento incremental del grado de definición de un sistema y su

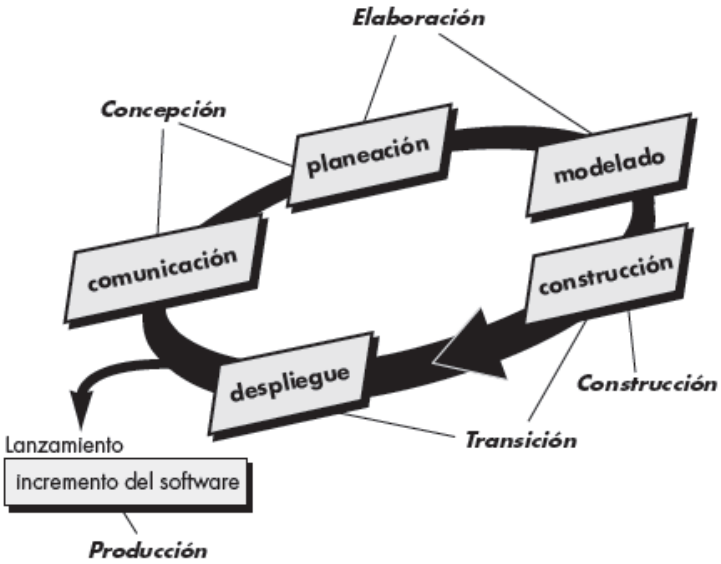
implementación, mientras que disminuye su grado de riesgo. La otra es un conjunto de puntos de referencia de anclaje puntual para asegurar el compromiso del participante con soluciones factibles y mutuamente satisfactorias.

Con el empleo del modelo espiral, el software se desarrolla en una serie de entregas evolutivas. Durante las primeras iteraciones, lo que se entrega puede ser un modelo o prototipo. En las iteraciones posteriores se producen versiones cada vez más completas del sistema cuya ingeniería se está haciendo.



2.1.5. El proceso unificado

En cierto modo, el proceso unificado es un intento por obtener los mejores rasgos y características de los modelos tradicionales del proceso del software, pero en forma que implemente muchos de los mejores principios del desarrollo ágil de software. El proceso unificado reconoce la importancia de la comunicación con el cliente y los métodos directos para describir su punto de vista respecto de un sistema (el caso de uso). Hace énfasis en la importancia de la arquitectura del software y “ayuda a que el arquitecto se centre en las metas correctas, tales como que sea comprensible, permita cambios futuros y la reutilización”. Se Sugiere un flujo del proceso iterativo e incremental, lo que da la sensación evolutiva que resulta esencial en el desarrollo moderno del software.



La fase de concepción del PU agrupa actividades tanto de comunicación con el cliente como de planeación. Al colaborar con los participantes, se identifican los requerimientos del negocio, se propone una arquitectura aproximada para el sistema y se desarrolla un plan para la naturaleza iterativa e incremental del proyecto en cuestión. Los requerimientos fundamentales del negocio se describen por medio de un conjunto de casos de uso preliminares que detallan las características y funciones que desea cada clase principal de usuarios. En este punto, la arquitectura no es más que un lineamiento tentativo de subsistemas principales y la función y rasgos que tienen. La arquitectura se mejorará después y se expandirá en un conjunto de modelos que representarán distintos puntos de vista del sistema. La planeación identifica los recursos, evalúa los riesgos principales, define un programa de actividades y establece una base para las fases que se van a aplicar a medida que avanza el incremento del software.

2.2. Métodos ágiles

Cualquier proceso del software ágil se caracteriza por la forma en la que aborda cierto número de suposiciones clave acerca de la mayoría de los proyectos de software:

1. Es difícil predecir qué requerimientos de software persistirán y cuáles cambiarán. También es difícil pronosticar cómo cambiarán las prioridades del cliente a medida que avanza el proyecto.

2. Para muchos tipos de software, el diseño y la construcción están imbricados. Es decir, ambas actividades deben ejecutarse en forma simultánea, de modo que los modelos de diseño se prueben a medida que se crean. Es difícil predecir cuánto diseño se necesita antes de que se use la construcción para probar el diseño.
3. El análisis, el diseño, la construcción y las pruebas no son tan predecibles como nos gustaría (desde un punto de vista de planeación).

La filosofía detrás de los métodos ágiles se refleja en el manifiesto ágil, que acordaron muchos de los desarrolladores líderes de estos métodos. Este manifiesto afirma: Estamos descubriendo mejores formas para desarrollar software, al hacerlo y al ayudar a otros a hacerlo. Gracias a este trabajo llegamos a valorar:

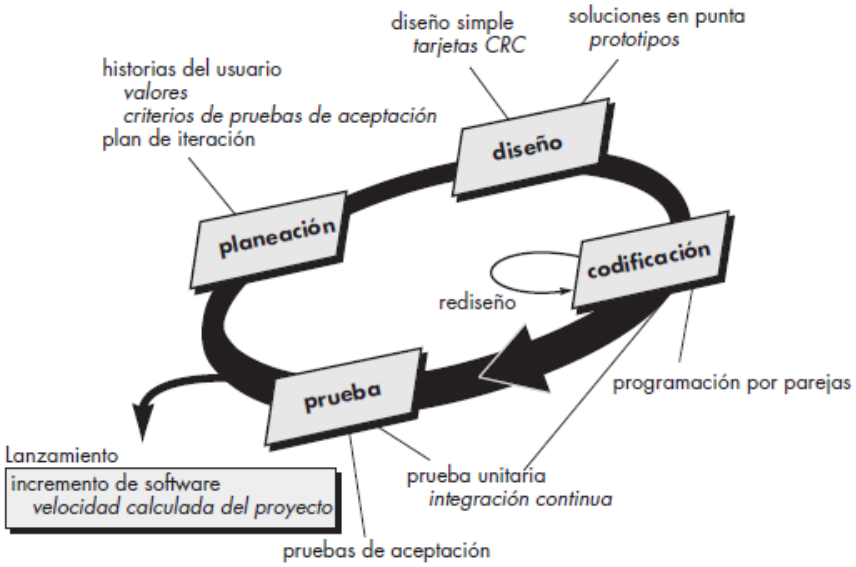
- A los individuos y las interacciones sobre los procesos y las herramientas
- Al software operativo sobre la documentación exhaustiva
- La colaboración con el cliente sobre la negociación del contrato
- La respuesta al cambio sobre el seguimiento de un plan

2.2.1. Principios de agilidad

Principio	Descripción
Participación del cliente	Los clientes deben intervenir estrechamente durante el proceso de desarrollo. Su función consiste en ofrecer y priorizar nuevos requerimientos del sistema y evaluar las iteraciones del mismo.
Entrega incremental	El software se desarrolla en incrementos y el cliente especifica los requerimientos que se van a incluir en cada incremento.
Personas, no procesos	Tienen que reconocerse y aprovecharse las habilidades del equipo de desarrollo. Debe permitirse a los miembros del equipo desarrollar sus propias formas de trabajar sin procesos establecidos.
Adoptar el cambio	Esperar a que cambien los requerimientos del sistema y, de este modo, diseñar el sistema para adaptar dichos cambios.
Mantener simplicidad	Enfocarse en la simplicidad tanto en el software a desarrollar como en el proceso de desarrollo. Siempre que sea posible, trabajar de manera activa para eliminar la complejidad del sistema.

2.2.2. El proceso XP

La programación extrema usa un enfoque orientado a objetos (véase el apéndice 2) como paradigma preferido de desarrollo, y engloba un conjunto de reglas y prácticas que ocurren en el contexto de cuatro actividades estructurales: planeación, diseño, codificación y pruebas. En la siguiente figura se ilustra el proceso XP y resalta algunas de las ideas y tareas clave que se asocian con cada actividad estructural. En los párrafos que siguen se resumen las actividades de XP clave.



Planeación. La actividad de planeación (también llamada juego de planeación) comienza escuchando – actividad para recabar requerimientos que permite que los miembros técnicos del equipo XP entiendan el contexto del negocio para el software y adquieran la sensibilidad de la salida y características principales y funcionalidad que se requieren—. Escuchar lleva a la creación de algunas "historias" (también llamadas historias del usuario) que describen la salida necesaria, características y funcionalidad del software que se va a elaborar

Diseño. El diseño XP sigue rigurosamente el principio MS (mantenlo sencillo). Un diseño sencillo siempre se prefiere sobre una representación más compleja. Además, el diseño guía la implementación de una historia conforme se escribe: nada más y nada menos. Se desalienta el diseño de funcionalidad adicional porque el desarrollador supone que se requerirá después. XP estimula el uso de las tarjetas CRC como un mecanismo eficaz para pensar en el software en un

contexto orientado a objetos. Las tarjetas CRC (clase-responsabilidad-colaborador) identifican y organizan las clases orientadas a objetos7 que son relevantes para el incremento actual de software.

Codificación. Después de que las historias han sido desarrolladas y de que se ha hecho el trabajo de diseño preliminar, el equipo no inicia la codificación, sino que desarrolla una serie de pruebas unitarias a cada una de las historias que se van a incluir en la entrega en curso (incremento de software).8 Una vez creada la prueba unitaria,9 el desarrollador está mejor capacitado para centrarse en lo que debe implementarse para pasar la prueba. No se agrega nada extraño (MS). Una vez que el código está terminado, se le aplica de inmediato una prueba unitaria, con lo que se obtiene retroalimentación instantánea para los desarrolladores.

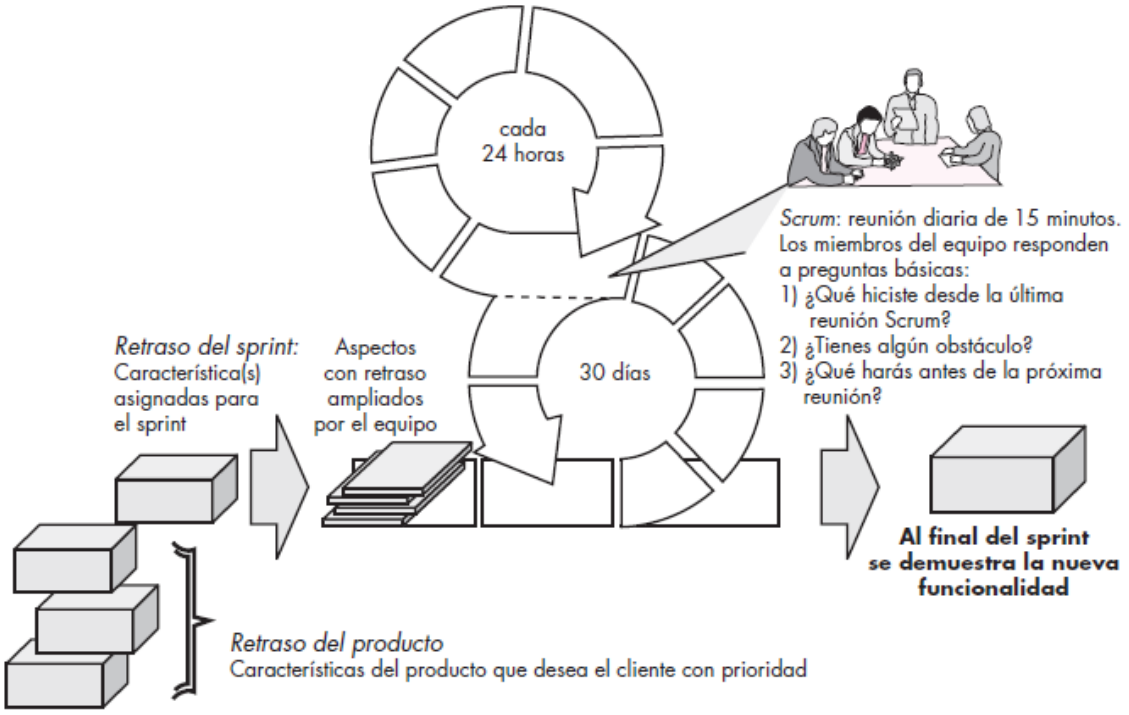
Un concepto clave durante la actividad de codificación (y uno de los aspectos del que más se habla en la XP) es la programación por parejas. XP recomienda que dos personas trabajen juntas en una estación de trabajo con el objeto de crear código para una historia. Esto da un mecanismo para la solución de problemas en tiempo real (es frecuente que dos cabezas piensen más que una) y para el aseguramiento de la calidad también en tiempo real (el código se revisa conforme se crea).

Pruebas. Ya se dijo que la creación de pruebas unitarias antes de que comience la codificación es un elemento clave del enfoque de XP. Las pruebas unitarias que se crean deben implementarse con el uso de una estructura que permita automatizarlas (de modo que puedan ejecutarse en repetidas veces y con facilidad). Esto estimula una estrategia de pruebas de regresión siempre que se modifique el código (lo que ocurre con frecuencia, dada la filosofía del rediseño en XP).

2.2.3. Scrum

Los principios Scrum son congruentes con el manifiesto ágil y se utilizan para guiar actividades de desarrollo dentro de un proceso de análisis que incorpora las siguientes actividades estructurales: requerimientos, análisis, diseño, evolución y entrega. Dentro de cada actividad estructural, las tareas del trabajo

ocurren con un patrón del proceso (que se estudia en el párrafo siguiente) llamado sprint. El trabajo realizado dentro de un sprint (el número de éstos que requiere cada actividad estructural variará en función de la complejidad y tamaño del producto) se adapta al problema en cuestión y se define —y con frecuencia se modifica— en tiempo real por parte del equipo Scrum.



Scrum acentúa el uso de un conjunto de patrones de proceso del software que han demostrado ser eficaces para proyectos con plazos de entrega muy apretados, requerimientos cambiantes y negocios críticos. Cada uno de estos patrones de proceso define un grupo de acciones de desarrollo:

Retraso: lista de prioridades de los requerimientos o características del proyecto que dan al cliente un valor del negocio. Es posible agregar en cualquier momento otros aspectos al retraso (ésta es la forma en la que se introducen los cambios). El gerente del proyecto evalúa el retraso y actualiza las prioridades según se requiera.

Sprints: consiste en unidades de trabajo que se necesitan para alcanzar un requerimiento definido en el retraso que debe ajustarse en una caja de tiempo14 predefinida (lo común son 30 días). Durante el sprint no se introducen cambios

(por ejemplo, aspectos del trabajo retrasado). Así, el sprint permite a los miembros del equipo trabajar en un ambiente de corto plazo, pero estable.

Reuniones Scrum: son reuniones breves (de 15 minutos, por lo general) que el equipo Scrum efectúa a diario. Hay tres preguntas clave que se pide que respondan todos los miembros del equipo:

- ¿Qué hiciste desde la última reunión del equipo?
- ¿Qué obstáculos estás encontrando?
- ¿Qué planeas hacer mientras llega la siguiente reunión del equipo?

Scrum Master: Un líder del equipo, llamado maestro Scrum, dirige la junta y evalúa las respuestas de cada persona. La junta Scrum ayuda al equipo a descubrir los problemas potenciales tan pronto como sea posible. Asimismo, estas juntas diarias llevan a la “socialización del conocimiento” con lo que se promueve una estructura de equipo con organización propia

Demostraciones preliminares: entregar el incremento de software al cliente de modo que la funcionalidad que se haya implementado pueda demostrarse al cliente y éste pueda evaluarla

Capítulo III: Ingeniería de Requerimientos

Ingeniería de Requerimientos

3.1. Análisis de los requerimientos

El análisis de los requerimientos da como resultado la especificación de las características operativas del software, indica la interfaz de éste y otros elementos del sistema, y establece las restricciones que limitan al software. El análisis de los requerimientos permite al profesional (sin importar si se llama ingeniero de software, analista o modelista) construir sobre los requerimientos básicos establecidos durante las tareas de concepción, indagación y negociación, que son parte de la ingeniería de los requerimientos.

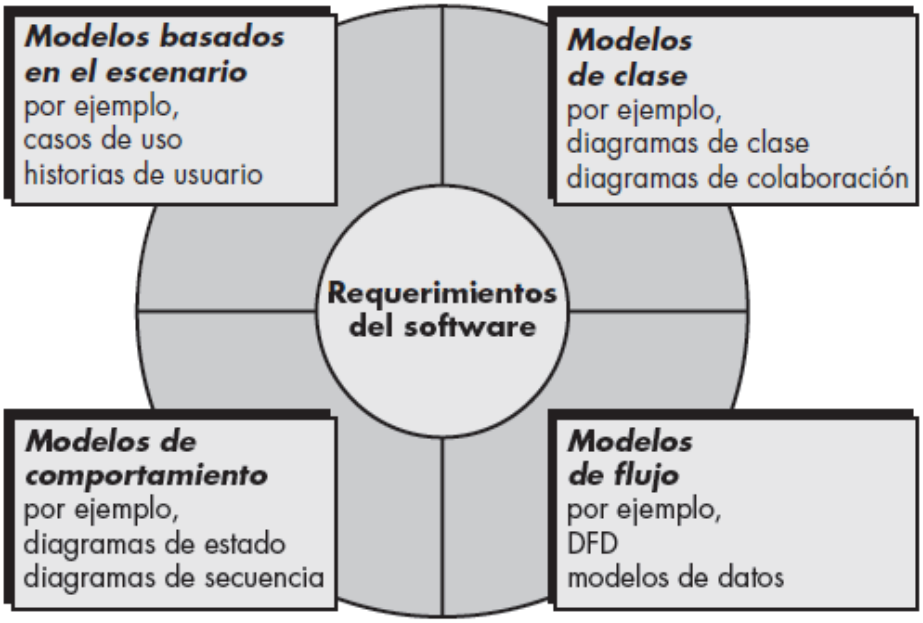
La acción de modelar los requerimientos da como resultado uno o más de los siguientes tipos

de modelo:

- Modelos basados en el escenario de los requerimientos desde el punto de vista de distintos “actores” del sistema.
- Modelos de datos, que ilustran el dominio de información del problema.
- Modelos orientados a clases, que representan clases orientadas a objetos (atributos y

operaciones) y la manera en la que las clases colaboran para cumplir con los requerimientos del sistema.

- Modelos orientados al flujo, que representan los elementos funcionales del sistema y la manera como transforman los datos a medida que se avanza a través del sistema.
- Modelos de comportamiento, que ilustran el modo en el que se comparte el software como consecuencia de “eventos” externos.



Durante el modelado de los requerimientos, la atención se centra en qué, no en cómo. ¿Qué interacción del usuario ocurre en una circunstancia particular?, ¿qué objetos manipula el sistema?, ¿qué funciones debe realizar el sistema?, ¿qué comportamientos tiene el sistema?, ¿qué interfaces se definen? y ¿qué restricciones son aplicables?

En esta etapa tal vez no fuera posible tener la especificación completa de los requerimientos. El cliente quizá no esté seguro de qué es lo que requiere con precisión para ciertos aspectos del sistema. Puede ser que el desarrollador esté inseguro de que algún enfoque específico cumpla de manera apropiada la función y el desempeño.

Estas realidades hablan a favor de un enfoque iterativo para el análisis y el modelado de los requerimientos. El analista debe modelar lo que se sabe y usar el modelo como base para el diseño del incremento del software.

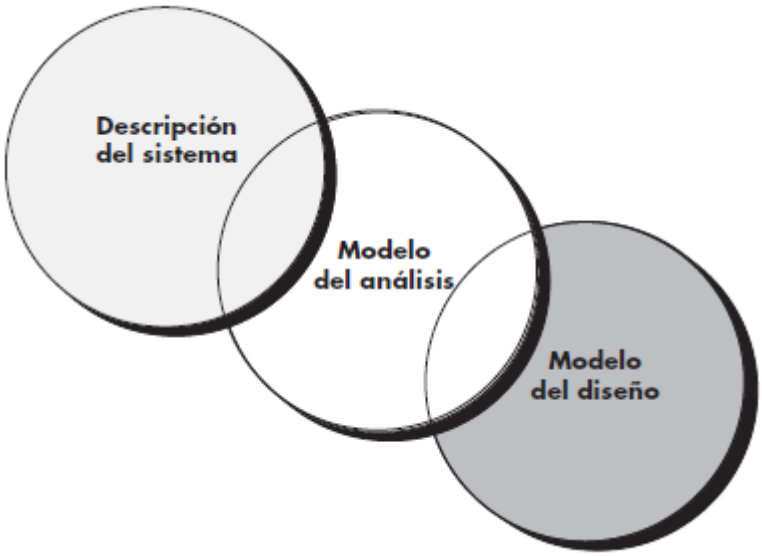
El modelo de requerimientos debe lograr tres objetivos principales:

1. Describir lo que requiere el cliente,
2. Establecer una base para la creación de un diseño de software y
3. Definir un conjunto de requerimientos que puedan validarse una vez construido el software.

El modelo de análisis es un puente entre la descripción en el nivel del sistema que se centra en éste en lo general o en la funcionalidad del negocio que se logra con la aplicación de software, hardware,

datos, personas y otros elementos del sistema y un diseño de software (véanse los capítulos 8 a 13) que describa la arquitectura de la aplicación del software, la interfaz del usuario y la estructura

en el nivel del componente. Esta relación se ilustra en la siguiente figura



3.2. Los requerimientos del usuario y del sistema

Los requerimientos para un sistema son descripciones de lo que el sistema debe hacer: el servicio que ofrece y las restricciones en su operación. Tales requerimientos reflejan las necesidades de los clientes por un sistema que atienda cierto propósito, como sería controlar un dispositivo, colocar un pedido o buscar información. Al proceso de descubrir, analizar, documentar y verificar estos servicios y restricciones se le llama ingeniería de requerimientos (IR).

El término “requerimiento” no se usa de manera continua en la industria del software. En algunos casos, un requerimiento es simplemente un enunciado abstracto de alto nivel en un servicio que debe proporcionar un sistema, o bien,

una restricción sobre un sistema. En el otro extremo, consiste en una definición detallada y formal de una función del sistema. Davis (1993) explica por qué existen esas diferencias:

Si una compañía desea otorgar un contrato para un gran proyecto de desarrollo de software, tiene que definir sus necesidades de una forma suficientemente abstracta para que una solución no esté predefinida. Los requerimientos deben redactarse de tal forma que muchos proveedores liciten en pos del contrato, ofreciendo, tal vez, diferentes maneras de cubrir las necesidades de organización del cliente. Una vez otorgado el contrato, el proveedor tiene que escribir con más detalle una definición del sistema para el cliente, de modo que éste comprenda y valide lo que hará el software. Estos documentos suelen nombrarse documentos de requerimientos para el sistema.

Algunos de los problemas que surgen durante el proceso de ingeniería de requerimientos son resultado del fracaso de hacer una separación clara entre esos diferentes niveles de descripción. En este texto se distinguen con el uso del término “requerimientos del usuario” para representar los requerimientos abstractos de alto nivel; y “requerimientos del sistema” para caracterizar la descripción detallada de lo que el sistema debe hacer. Los requerimientos del usuario y los requerimientos del sistema se definen del siguiente modo:

1. **Los requerimientos del usuario** son enunciados, en un lenguaje natural junto con diagramas, acerca de qué servicios esperan los usuarios del sistema, y de las restricciones con las cuales éste debe operar.

2. **Los requerimientos del sistema** son descripciones más detalladas de las funciones, los servicios y las restricciones operacionales del sistema de software. El documento de requerimientos del sistema (llamado en ocasiones especificación funcional) tiene que definir con exactitud lo que se implementará. Puede formar parte del contrato entre el comprador del sistema y los desarrolladores del software.

En el siguiente ejemplo se observa que el requerimiento del usuario es muy general. Los requerimientos del sistema ofrecen información más específica sobre los servicios y las funciones del sistema que se implementará.

Ejemplo:

Requerimiento del usuario

1. El MHC-PMS elaborará mensualmente informes administrativos que revelen el costo de los medicamentos prescritos por cada clínica durante ese mes.

Requerimientos del sistema

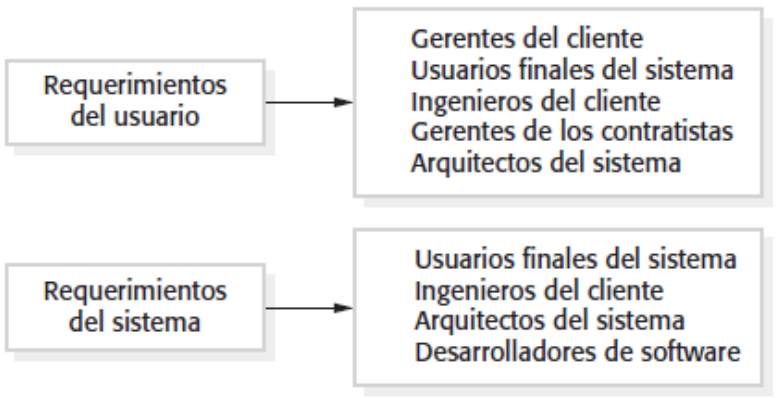
1.1 En el último día laboral de cada mes se redactará un resumen de los medicamentos prescritos, su costo y las clínicas que los prescriben.

1.2 El sistema elaborará automáticamente el informe que se imprimirá después de las 17:30 del último día laboral del mes.

1.3 Se realizará un reporte para cada clínica junto con los nombres de cada medicamento, el número de prescripciones, las dosis prescritas y el costo total de los medicamentos prescritos.

1.4 Si los medicamentos están disponibles en diferentes unidades de dosis (por ejemplo, 10 mg, 20 mg) se harán informes por separado para cada unidad de dosis.

1.5 El acceso a los informes de costos se restringirá.



3.3. Requerimientos funcionales y no funcionales

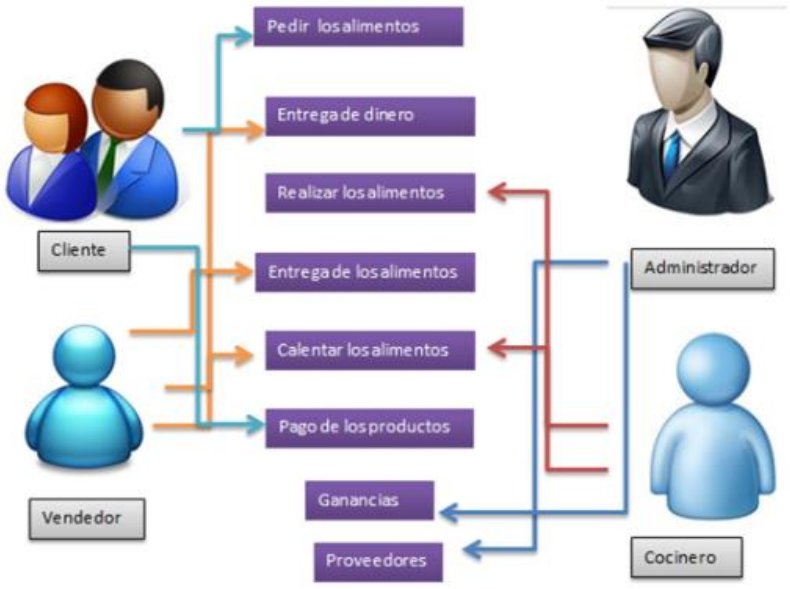
1. **Requerimientos funcionales** Son enunciados acerca de servicios que el sistema debe proveer, de cómo debería reaccionar el sistema a entradas particulares y de cómo debería comportarse el sistema en situaciones

específicas. En algunos casos, los requerimientos funcionales también explican lo que no debe hacer el sistema.

2. **Requerimientos no funcionales** Son limitaciones sobre servicios o funciones que ofrece el sistema. Incluyen restricciones tanto de temporización y del proceso de desarrollo, como impuestas por los estándares. Los requerimientos no funcionales se suelen aplicar al sistema como un todo, más que a características o a servicios individuales del sistema.

Los requerimientos funcionales para un sistema refieren lo que el sistema debe hacer. Tales requerimientos dependen del tipo de software que se esté desarrollando, de los usuarios esperados del software y del enfoque general que adopta la organización cuando se escriben los requerimientos. Al expresarse como requerimientos del usuario, los requerimientos funcionales se describen por lo general de forma abstracta que entiendan los usuarios del sistema. Sin embargo, requerimientos funcionales más específicos del sistema detallan las funciones del sistema, sus entradas y salidas, sus excepciones, etcétera.

Ejemplo:



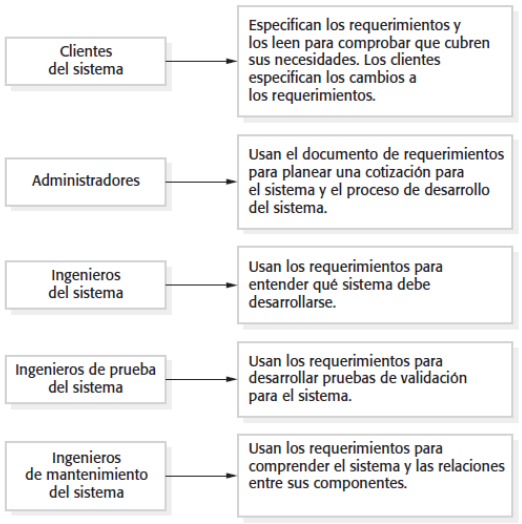
Los requerimientos no funcionales, como indica su nombre, son requerimientos que no se relacionan directamente con los servicios específicos que el sistema entrega a sus usuarios. Pueden relacionarse con propiedades emergentes del sistema, como fiabilidad, tiempo de respuesta y uso de almacenamiento. De forma alternativa, pueden definir restricciones sobre la implementación del sistema, como las capacidades de los dispositivos I/O o las representaciones de datos usados en las interfaces con otros sistemas. Los requerimientos no funcionales, como el rendimiento, la seguridad o la disponibilidad, especifican o restringen por lo general características del sistema como un todo.

Los requerimientos no funcionales pueden agruparse por requerimientos no funcionales del producto, requerimientos no funcionales organizacionales y requerimientos no funcionales externos. A continuación, se detallan estos requerimientos no funcionales.

3.4. El documento de requerimientos de software

El documento de requerimientos de software (llamado algunas veces especificación de requerimientos de software o SRS) es un comunicado oficial de lo que deben implementar los desarrolladores del sistema. Incluye tanto los requerimientos del usuario para un sistema, como una especificación detallada de los requerimientos del sistema. En ocasiones, los requerimientos del usuario y del sistema se integran en una sola descripción. En otros casos, los requerimientos del usuario se definen en una introducción a la especificación de requerimientos del sistema. Si hay un gran número de requerimientos, los requerimientos del sistema detallados podrían presentarse en un documento aparte.

Son esenciales los documentos de requerimientos cuando un contratista externo diseña el sistema de software. Sin embargo, los métodos de desarrollo ágiles argumentan que los requerimientos cambian tan rápidamente que un documento de requerimientos se vuelve obsoleto tan pronto como se escribe, así que el esfuerzo se desperdicia en gran medida. En lugar de un documento formal, los enfoques como la programación extrema (Beck, 1999) recopilan de manera incremental requerimientos del usuario y los escriben en tarjetas como historias de usuario. De esa manera, el usuario da prioridad a los requerimientos para su implementación en el siguiente incremento del sistema. La siguiente figura tomada del libro del autor con Gerald Kotonya sobre ingeniería de requerimientos (Kotonya y Sommerville, 1998), muestra a los posibles usuarios del documento y cómo ellos lo utilizan.



La siguiente figura indica una posible organización para un documento de requerimientos basada en un estándar del IEEE para documentos de requerimientos (IEEE, 1998). Este estándar es genérico y se adapta a usos específicos.

Capítulo	Descripción
Prefacio	Debe definir el número esperado de lectores del documento, así como describir su historia de versiones, incluidas las causas para la creación de una nueva versión y un resumen de los cambios realizados en cada versión.
Introducción	Describe la necesidad para el sistema. Debe detallar brevemente las funciones del sistema y explicar cómo funcionará con otros sistemas. También tiene que indicar cómo se ajusta el sistema en los objetivos empresariales o estratégicos globales de la organización que comisiona el software.
Glosario	Define los términos técnicos usados en el documento. No debe hacer conjeturas sobre la experiencia o la habilidad del lector.
Definición de requerimientos del usuario	Aquí se representan los servicios que ofrecen al usuario. También, en esta sección se describen los requerimientos no funcionales del sistema. Esta descripción puede usar lenguaje natural, diagramas u otras observaciones que sean comprensibles para los clientes. Deben especificarse los estándares de producto y proceso que tienen que seguirse.
Arquitectura del sistema	Este capítulo presenta un panorama de alto nivel de la arquitectura anticipada del sistema, que muestra la distribución de funciones a través de los módulos del sistema. Hay que destacar los componentes arquitectónicos que sean de reutilización.
Especificación de requerimientos del sistema	Debe representar los requerimientos funcionales y no funcionales con más detalle. Si es preciso, también pueden detallarse más los requerimientos no funcionales. Pueden definirse las interfaces a otros sistemas.
Modelos del sistema	Pueden incluir modelos gráficos del sistema que muestren las relaciones entre componentes del sistema, el sistema y su entorno. Ejemplos de posibles modelos son los modelos de objeto, modelos de flujo de datos o modelos de datos semánticos.
Evolución del sistema	Describe los supuestos fundamentales sobre los que se basa el sistema, y cualquier cambio anticipado debido a evolución de hardware, cambio en las necesidades del usuario, etc. Esta sección es útil para los diseñadores del sistema, pues los ayuda a evitar decisiones de diseño que restringirían probablemente futuros cambios al sistema.
Apéndices	Brindan información específica y detallada que se relaciona con la aplicación a desarrollar; por ejemplo, descripciones de hardware y bases de datos. Los requerimientos de hardware definen las configuraciones, mínima y óptima, del sistema. Los requerimientos de base de datos delimitan la organización lógica de los datos usados por el sistema y las relaciones entre datos.
Índice	Pueden incluirse en el documento varios índices. Así como un índice alfabético normal, uno de diagramas, un índice de funciones, etcétera.

3.5. Especificación de requerimientos

La especificación de requerimientos es el proceso de escribir, en un documento de requerimientos, los requerimientos del usuario y del sistema. De manera ideal, los requerimientos del usuario y del sistema deben ser claros, sin ambigüedades,

fáciles de entender, completos y consistentes. Esto en la práctica es difícil de lograr, pues los participantes interpretan los requerimientos de formas diferentes y con frecuencia en los requerimientos hay conflictos e inconsistencias inherentes.

Los requerimientos del usuario para un sistema deben describir los requerimientos funcionales y no funcionales, de forma que sean comprensibles para los usuarios del sistema que no cuentan con un conocimiento técnico detallado. De manera ideal, deberían especificar sólo el comportamiento externo del sistema. El documento de requerimientos no debe incluir detalles de la arquitectura o el diseño del sistema. En consecuencia, si usted escribe los requerimientos del usuario, no tiene que usar jerga de software, anotaciones estructuradas o formales. Debe escribir los requerimientos del usuario en lenguaje natural, con tablas y formas sencillas, así como diagramas intuitivos.

Los requerimientos del usuario se escriben casi siempre en lenguaje natural, complementado con diagramas y tablas adecuados en el documento de requerimientos. Los requerimientos del sistema se escriben también en lenguaje natural, pero de igual modo se utilizan otras notaciones basadas en formas, modelos gráficos del sistema o modelos matemáticos del sistema. La figura siguiente resume las posibles anotaciones que podrían usarse para escribir requerimientos del sistema.

Notación	Descripción
Enunciados en lenguaje natural	Los requerimientos se escriben al usar enunciados numerados en lenguaje natural. Cada enunciado debe expresar un requerimiento.
Lenguaje natural estructurado	Los requerimientos se escriben en lenguaje natural en una forma o plantilla estándar. Cada campo ofrece información de un aspecto del requerimiento.
Lenguajes de descripción de diseño	Este enfoque usa un lenguaje como un lenguaje de programación, pero con características más abstractas para especificar los requerimientos al definir un modelo operacional del sistema. Aunque en la actualidad este enfoque se usa raras veces, aún tiene utilidad para especificaciones de interfaz.
Anotaciones gráficas	Los modelos gráficos, complementados con anotaciones de texto, sirven para definir los requerimientos funcionales del sistema; los casos de uso del UML y los diagramas de secuencia se emplean de forma común.
Especificaciones matemáticas	Dichas anotaciones se basan en conceptos matemáticos como máquinas o conjuntos de estado finito. Aunque tales especificaciones sin ambigüedades pueden reducir la imprecisión en un documento de requerimientos, la mayoría de los clientes no comprenden una especificación formal. No pueden comprobar que representa lo que quieren y por ello tienen reticencia para aceptarlo como un contrato de sistema.

3.6. Descubrimiento de requerimientos

El descubrimiento de requerimientos (llamado a veces adquisición de requerimientos) es el proceso de recopilar información sobre el sistema requerido y los sistemas existentes, así como de separar, a partir de esta información, los requerimientos del usuario y del sistema. Las fuentes de información durante la fase de descubrimiento de requerimientos incluyen documentación, participantes del sistema y especificaciones de sistemas similares. La interacción con los participantes es a través de entrevistas y observaciones, y pueden usarse escenarios y prototipos para ayudar a los participantes a entender cómo será el sistema.

Entrevistas

Las entrevistas formales o informales con participantes del sistema son una parte de la

mayoría de los procesos de ingeniería de requerimientos. En estas entrevistas, el equipo de ingeniería de requerimientos formula preguntas a los participantes sobre el sistema que actualmente usan y el sistema que se va a desarrollar. Los requerimientos se derivan de las respuestas a dichas preguntas. Las entrevistas son de dos tipos:

1. Entrevistas cerradas, donde los participantes responden a un conjunto de preguntas preestablecidas.
2. Entrevistas abiertas, en las cuales no hay agenda predefinida. El equipo de ingeniería de requerimientos explora un rango de conflictos con los participantes del sistema y, como resultado, desarrolla una mejor comprensión de sus necesidades.

Escenarios

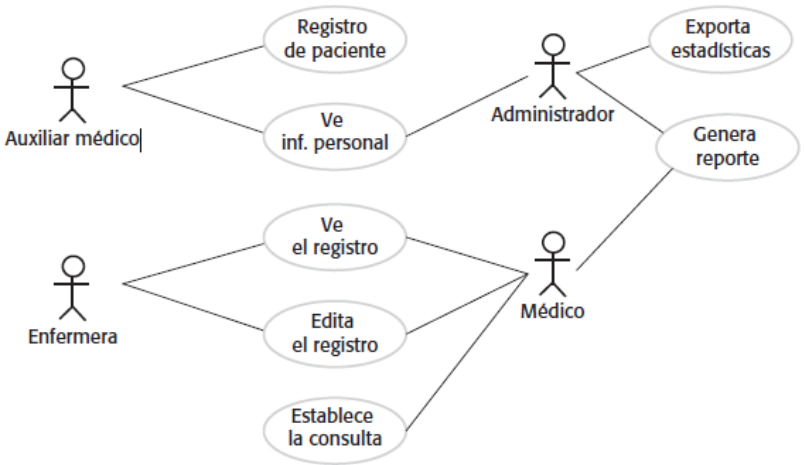
Por lo general, las personas encuentran más sencillo vincularse con ejemplos reales que con descripciones abstractas. Pueden comprender y criticar un escenario sobre cómo interactuar con un sistema de software. Los ingenieros de requerimientos usan la información obtenida de esta discusión para formular los verdaderos requerimientos del sistema.

Los escenarios son particularmente útiles para detallar un bosquejo de descripción de requerimientos. Se trata de ejemplos sobre descripciones de sesiones de interacción. Cada escenario abarca comúnmente una interacción o un número pequeño de interacciones posibles.

Un escenario comienza con un bosquejo de la interacción. Durante el proceso de adquisición, se suman detalles a éste para crear una representación completa de dicha interacción. En su forma más general, un escenario puede incluir:

Casos de uso

Los casos de uso se documentan con el empleo de un diagrama de caso de uso de alto nivel. El conjunto de casos de uso representa todas las interacciones posibles que se describirán en los requerimientos del sistema. Los actores en el proceso, que pueden ser individuos u otros sistemas, se representan como figuras sencillas. Cada clase de interacción se constituye como una elipse con etiqueta. Líneas vinculan a los actores con la interacción. De manera opcional, se agregan puntas de flecha a las líneas para mostrar cómo se inicia la interacción.



Etnografía

Los sistemas de software no existen de forma aislada; se utilizan en un contexto social y organizacional, y los requerimientos de sistemas de software se derivan y se restringen acorde a ese contexto. Satisfacer esos requerimientos sociales y organizacionales es crítico para el éxito del sistema. Una razón de por qué

muchos sistemas de software se entregan, pero nunca se utilizan es porque no se toma en cuenta la importancia de este tipo de requerimientos.

La etnografía es una técnica de observación que se puede utilizar para entender los requerimientos sociales y organizacionales. Un analista se sumerge por sí solo en el entorno laboral donde el sistema se utilizará. El trabajo diario se observa y se hacen notas de las tareas reales en las que los participantes están involucrados. La etnografía es especialmente efectiva para descubrir dos tipos de requerimientos:

1. Los requerimientos que se derivan de la forma en la que la gente trabaja realmente más que de la forma en la que las definiciones de los procesos establecen que debería trabajar.
2. Los requerimientos que se derivan de la cooperación y conocimiento de las actividades de la gente.

3.7. Validación de requerimientos

La validación de requerimientos es el proceso de verificar que los requerimientos definan realmente el sistema que en verdad quiere el cliente. Se traslapa con el análisis, ya que se interesa por encontrar problemas con los requerimientos. La validación de requerimientos es importante porque los errores en un documento de requerimientos pueden conducir a grandes costos por tener que rehacer, cuando dichos problemas se descubren durante el desarrollo del sistema o después de que éste se halla en servicio.

En general, el costo por corregir un problema de requerimientos al hacer un cambio en el sistema es mucho mayor que reparar los errores de diseño o codificación. La razón es que un cambio a los requerimientos significa generalmente que también deben cambiar el diseño y la implementación del sistema.

Durante el proceso de validación de requerimientos, tienen que realizarse diferentes tipos de comprobaciones sobre los requerimientos contenidos en el documento de requerimientos. Dichas comprobaciones incluyen:

1. **Comprobaciones de validez** Un usuario quizá crea que necesita un sistema para realizar ciertas funciones. Sin embargo, con mayor consideración y análisis se logra identificar las funciones adicionales o diferentes que se requieran. Los sistemas tienen diversos participantes con diferentes necesidades, y cualquier conjunto de requerimientos es inevitablemente un compromiso a través de la comunidad de participantes.
2. **Comprobaciones de consistencia** Los requerimientos en el documento no deben estar en conflicto. Esto es, no debe haber restricciones contradictorias o descripciones diferentes de la misma función del sistema.
3. **Comprobaciones de totalidad** El documento de requerimientos debe incluir requerimientos que definan todas las funciones y las restricciones pretendidas por el usuario del sistema.
4. **Comprobaciones de realismo** Al usar el conocimiento de la tecnología existente, los requerimientos deben comprobarse para garantizar que en realidad pueden implementarse. Dichas comprobaciones también tienen que considerar el presupuesto y la fecha para el desarrollo del sistema.
5. **Verificabilidad** Para reducir el potencial de disputas entre cliente y contratista, los requerimientos del sistema deben escribirse siempre de manera que sean verificables. Esto significa que usted debe ser capaz de escribir un conjunto de pruebas que demuestren que el sistema entregado cumpla cada requerimiento especificado.

3.8. Administración del cambio en los requerimientos

La administración del cambio en los requerimientos debe aplicarse a todos los cambios propuestos a los requerimientos de un sistema, después de aprobarse el documento de requerimientos. La administración del cambio es esencial porque es necesario determinar si los beneficios de implementar nuevos requerimientos están justificados por los costos de la implementación. La ventaja de usar un proceso formal para la administración del cambio es que todas las

propuestas de cambio se tratan de manera consistente y los cambios al documento de requerimientos se realizan en una forma controlada.

Existen tres etapas principales de un proceso de administración del cambio:

1. **Análisis del problema y especificación del cambio** El proceso comienza con la identificación de un problema en los requerimientos o, en ocasiones, con una propuesta de cambio específica. Durante esta etapa, el problema o la propuesta de cambio se analizan para comprobar que es válida. Este análisis retroalimenta al solicitante del cambio, quien responderá con una propuesta de cambio de requerimientos más específica, o decidirá retirar la petición.

2. **Análisis del cambio y estimación del costo** El efecto del cambio propuesto se valora usando información de seguimiento y conocimiento general de los requerimientos del sistema. El costo por realizar el cambio se estima en términos de modificaciones al documento de requerimientos y, si es adecuado, al diseño y la implementación del sistema. Una vez completado este análisis, se toma una decisión acerca de si se procede o no con el cambio de requerimientos.

3. **Implementación del cambio** Se modifican el documento de requerimientos y, donde sea necesario, el diseño y la implementación del sistema. Hay que organizar el documento de requerimientos de forma que sea posible realizar cambios sin reescritura o reorganización extensos. Conforme a los programas, la variabilidad en los documentos se logra al minimizar las referencias externas y al hacer las secciones del documento tan modulares como sea posible. De esta manera, secciones individuales pueden modificarse y sustituirse sin afectar otras partes del documento.

Capítulo IV: Modelado del Software

Modelado del Software

El modelado de software es una actividad importante dentro de la Ingeniería del Software, que consiste en la creación de modelos abstractos que representan el comportamiento y la estructura de un sistema de software.

Estos modelos se utilizan para comprender, especificar, diseñar, implementar, probar y mantener sistemas de software. Los modelos se crean utilizando diferentes notaciones y herramientas, como diagramas UML (Unified Modeling Language), diagramas de flujo de datos, diagramas de clases, entre otros.

El modelado de software es una actividad iterativa y se realiza a lo largo de todo el ciclo de vida del software. En la etapa de análisis, los modelos se utilizan para comprender los requisitos del sistema, mientras que en la etapa de diseño se utilizan para crear la arquitectura del sistema y especificar los componentes del software. En la etapa de implementación, los modelos se utilizan para generar código y en la etapa de prueba se utilizan para verificar que el software se comporta según lo especificado.

En resumen, el modelado de software es una actividad clave en la Ingeniería del Software, que permite crear modelos abstractos del sistema de software para entenderlo mejor, especificarlo, diseñarlo, implementarlo, probarlo y mantenerlo.

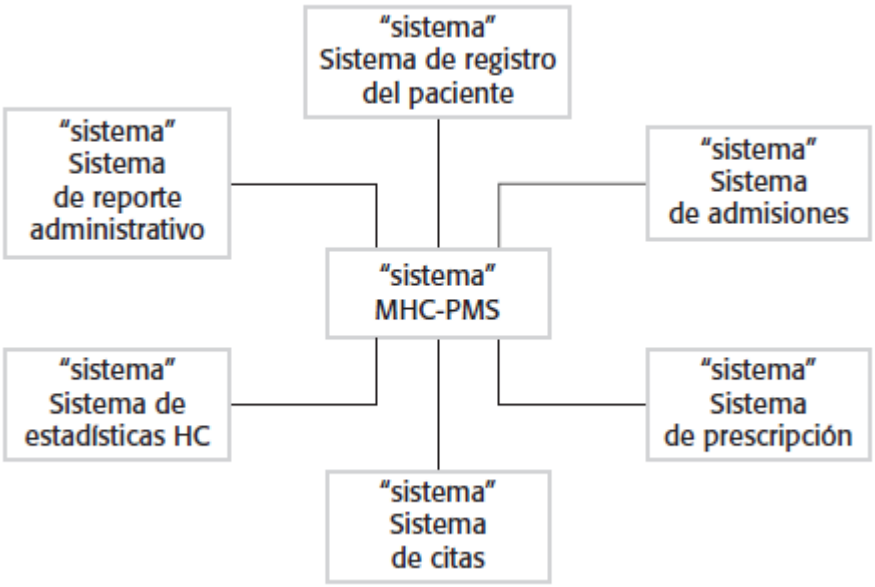
4.1. Modelos de contexto

En una primera etapa es la especificación de un sistema, debe decidir sobre las fronteras del sistema. Esto implica trabajar con los participantes del sistema para determinar cuál funcionalidad se incluirá en el sistema y cuál la ofrece el entorno del sistema.

Los modelos de contexto, por lo general, muestran que el entorno incluye varios sistemas automatizados. Por consiguiente, los modelos de contexto simples se usan junto con otros modelos, como los modelos de proceso empresarial. Éstos

describen procesos humanos y automatizados que se usan en sistemas particulares de software.

Ejemplo: El contexto del MHC-PMS



4.2. Diagramas de actividad

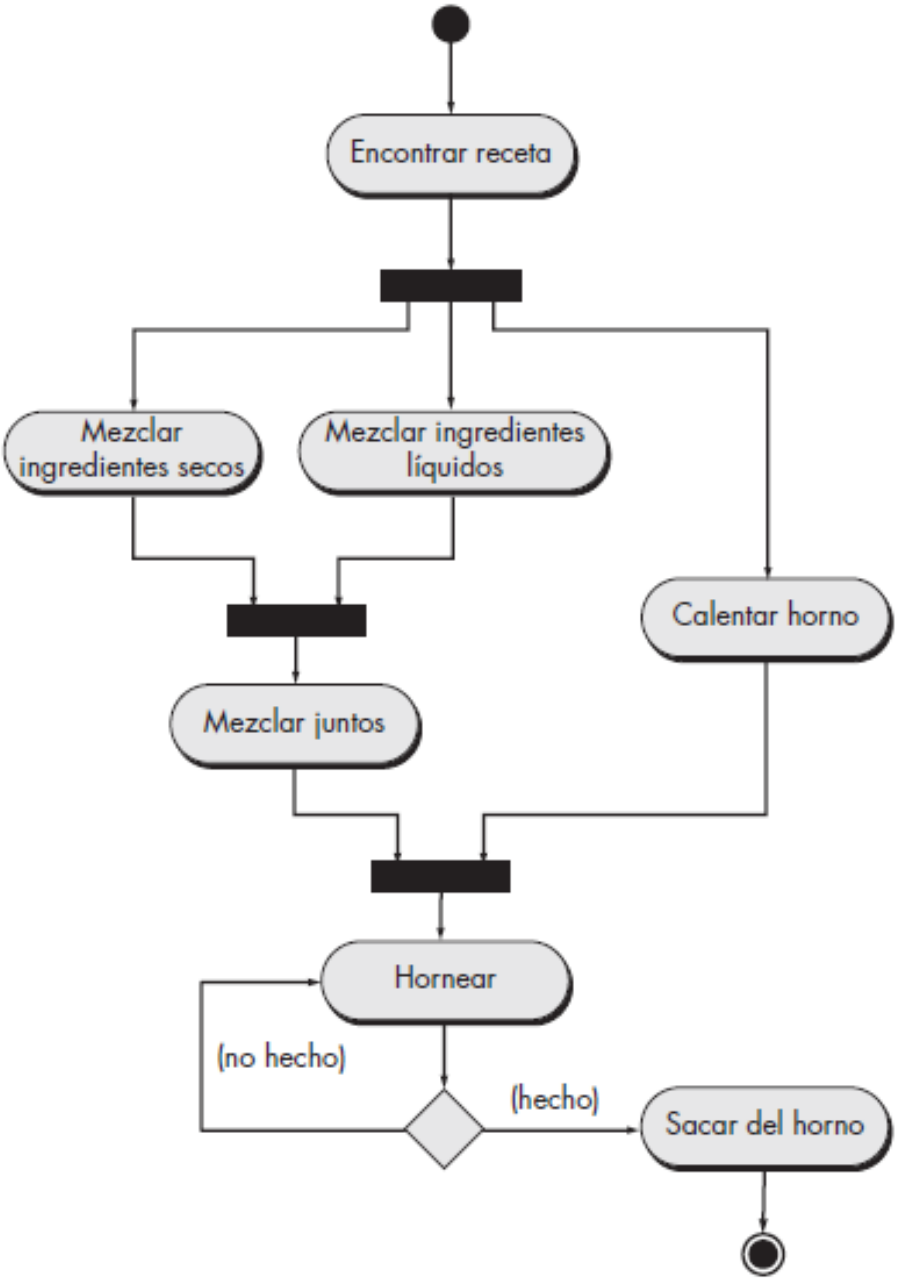
Un diagrama de actividad UML muestra el comportamiento dinámico de un sistema o de parte de un sistema a través del flujo de control entre acciones que realiza el sistema. Es similar a un diagrama de flujo, excepto porque un diagrama de actividad puede mostrar flujos concurrentes. El componente principal de un diagrama de actividad es un nodo acción, representado mediante un rectángulo redondeado, que corresponde a una tarea realizada por el sistema de software. Las flechas desde un nodo acción hasta otro indican el flujo de control; es decir, una flecha entre dos nodos acción significa que, después de completar la primera acción, comienza la segunda acción. Un punto negro sólido forma el nodo inicial que indica el punto de inicio de la actividad. Un punto negro rodeado por un círculo negro es el nodo final que indica el fin de la actividad.

Un *tenedor* (*fork*) representa la separación de actividades en dos o más actividades concurrentes. Se dibuja como una barra negra horizontal con una flecha apuntando hacia ella y dos o más flechas apuntando en sentido opuesto.

Una *unión* (*join*) es una forma de sincronizar flujos de control concurrentes. Se representa mediante una barra negra horizontal con dos o más flechas entrantes y una flecha saliente.

Un nodo de *decisión* corresponde a una rama en el flujo de control con base en una condición. Tal nodo se despliega como un triángulo blanco con una flecha entrante y dos o más flechas salientes. Cada flecha saliente se etiqueta con una guardia (una condición dentro de corchetes).

Ejemplo: Diagrama de actividad UML que muestra cómo hornear un pastel.

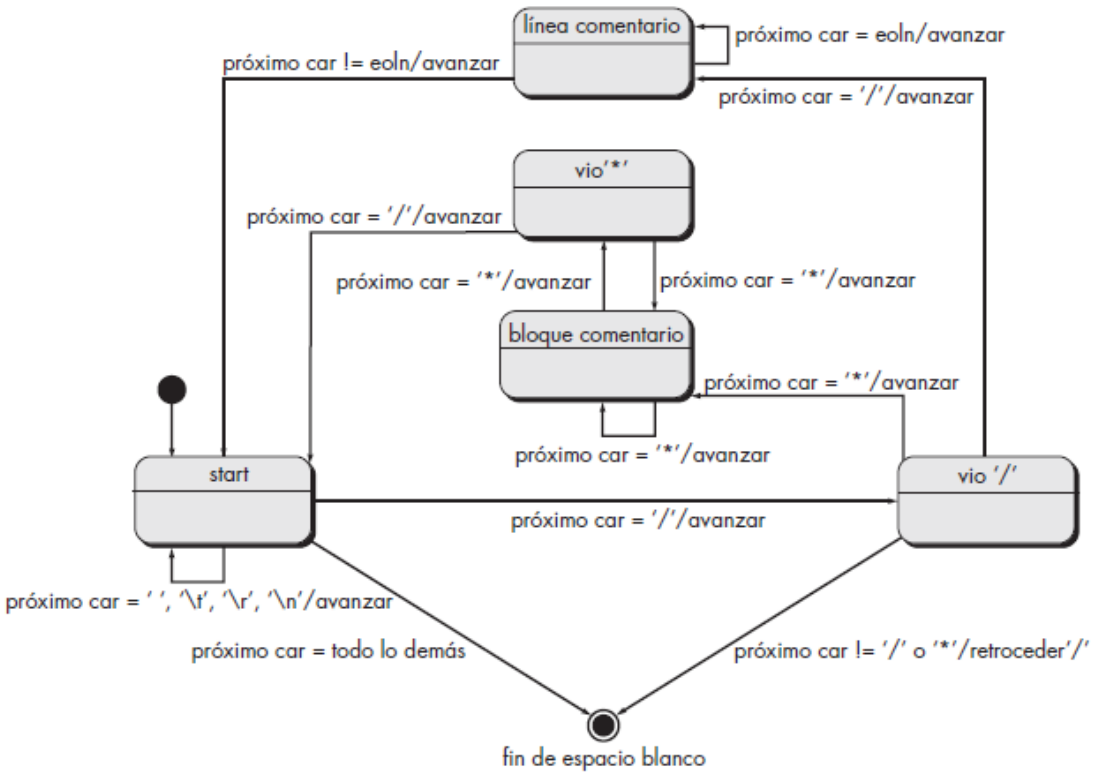


4.3. Diagramas de estado

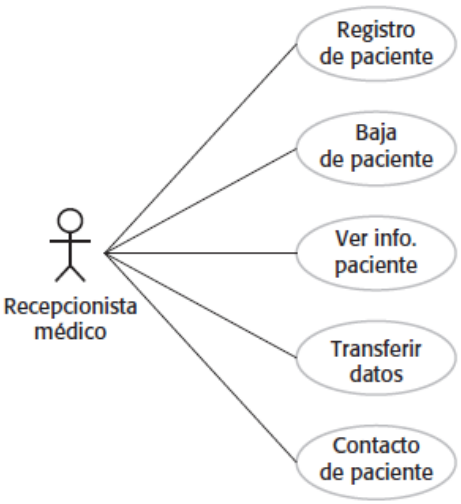
El comportamiento de un objeto en un punto particular en el tiempo con frecuencia depende del estado del objeto; es decir, de los valores de sus variables en dicho momento. Como ejemplo trivial, considere un objeto con una variable de instancia booleana. Cuando se pide realizar una operación, el objeto puede hacer una cosa si dicha variable es verdadera y hacer algo más si es falsa.

Un diagrama de estado UML modela los estados de un objeto, las acciones que se realizan dependiendo de dichos estados y las transiciones entre los estados del objeto.

Ejemplo: Diagrama de estado para avanzar sobre espacio blanco y comentarios en Java.



4.4. Modelado de casos de uso



El modelado de casos de uso fue desarrollado originalmente por Jacobson y sus colaboradores (1993) en la década de 1990, y se incorporó en el primer lanzamiento del UML (Rumbaugh et al., 1999). Como se estudió en el capítulo 4, el modelado de casos de uso se utiliza ampliamente para apoyar la adquisición de requerimientos. Un caso de uso puede tomarse como un simple escenario que describa lo que espera el usuario de un sistema. Cada caso de uso representa una tarea discreta que implica interacción externa con un sistema. En su forma más simple, un caso de uso se muestra como una elipse, con los actores que intervienen en el caso de uso representados como figuras humanas.

Los diagramas de caso de uso brindan un panorama bastante sencillo de una interacción, de modo que usted tiene que ofrecer más detalle para entender lo que está implicado. Este detalle puede ser una simple descripción textual, o una descripción estructurada en una tabla o un diagrama de secuencia, como se discute a continuación. Es posible elegir el formato más adecuado, dependiendo del caso de uso y del nivel de detalle que usted considere se requiera en el modelo. Para el autor, el formato más útil es un formato tabular estándar.

Elementos de un caso de uso:

- **Conjunto de secuencias de acciones**, cada secuencia representa un posible comportamiento del sistema.
- **Actores**, se tratan de los roles que pueden jugar los agentes que interactúan con el sistema. Los roles son jugados por personas, dispositivos, u otros sistemas. Podríamos distinguir entre actores primarios, para los cuales el objetivo del caso de uso es esencial y actores secundarios, que interactúan con el caso de uso, pero cuyo objetivo no es esencial.

- **Variantes**, son versiones especializadas, un caso de uso que extiende a otro o un caso de uso que incluye a otro

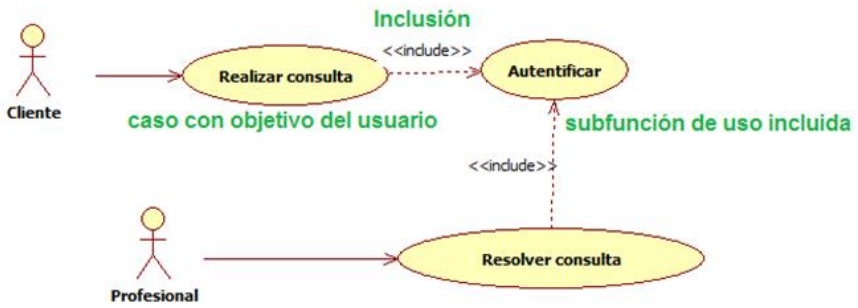
Tipos de relaciones en los diagramas de casos de uso

Comunicación: Relación (asociación) entre un actor y un caso de uso. El estereotipo de la relación de comunicación es: <<communicate>> aunque generalmente no se estipula ningún nombre, como podemos apreciar en el siguiente ejemplo de comunicación:

Inclusión: Un caso de uso base incorpora explícitamente el comportamiento de otro en algún lugar de su secuencia. La relación de inclusión sirve para enriquecer un caso de uso con otro y compartir una funcionalidad común entre varios casos de uso, también puede utilizarse para estructurar un caso de uso describiendo sus subfunciones. El caso de uso incluido existe únicamente con ese propósito, ya que no responde a un objetivo de un actor.

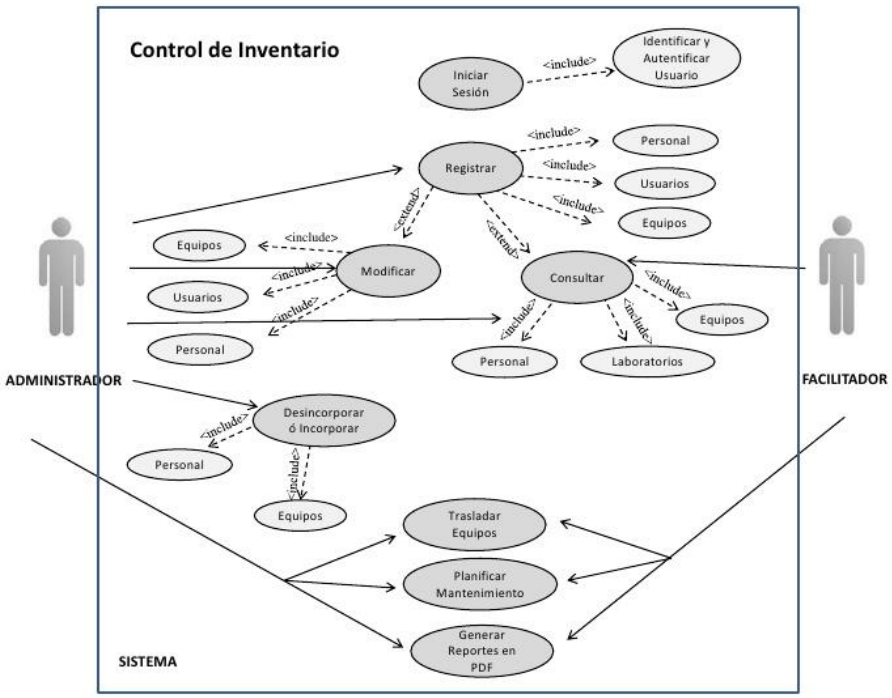
Estas relaciones se representan mediante una flecha discontinua con el estereotipo <<include>>. Algunos casos de uso típicos de inclusión son: comprobar, verificar, buscar, validar, autentificar o login… En principio, no deberíamos abusar de este tipo de relación, para no hacer una descomposición funcional del sistema.

Veamos un ejemplo de inclusión entre casos de uso:



Extensión: Un caso de uso base incorpora implícitamente el comportamiento de otro caso de uso en el lugar especificado indirectamente por este otro caso de uso. En el caso de uso base, la extensión se hace en una serie de puntos concretos y previstos en el momento del diseño, llamados puntos de extensión, los cuáles no son parte del flujo principal. La relación de extensión sirve para modelar: la parte opcional del sistema, un subflujo que sólo se ejecuta bajo ciertas condiciones o varios flujos que se pueden insertar en un punto determinado. Este tipo de relación produce confusión y no debería utilizarse en exceso. Conviene su uso sólo para insertar un nuevo comportamiento no previsto en un caso de uso existente. Estas relaciones se representan mediante una flecha discontinua con el estereotipo <<extend>>. Veamos un ejemplo de extensión:

Ejemplo de diagrama de casos de uso para un sistema de control de inventario



4.5. Diagramas de clase

Los diagramas de clase pueden usarse cuando se desarrolla un modelo de sistema orientado a objetos para mostrar las clases en un sistema y las asociaciones entre dichas clases. De manera holgada, una clase de objeto se considera como una definición general de un tipo de objeto del sistema. Una asociación es un vínculo entre clases, que indica que hay una relación entre dichas clases. En consecuencia, cada clase puede tener algún conocimiento de esta clase asociada.

Una clase está compuesta por tres elementos:

- Nombre de la clase
- Atributos
- Funciones.

Para representar la clase con estos elementos se utiliza una caja que es dividida en tres zonas utilizando para ello líneas horizontales:

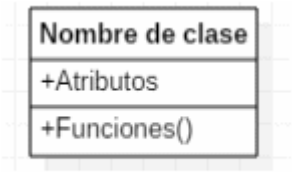
1. La primera de las zonas se utiliza para el nombre de la clase. En caso de que la clase sea abstracta se utilizará su nombre en cursiva.
2. La segunda de las zonas se utiliza para escribir los atributos de la clase, uno por línea y utilizando el siguiente formato:

   ```
   visibilidad nombre_atributo : tipo = valor-inicial { propiedades }
   ```

   Aunque esta es la forma “oficial” de escribirlas, es común simplificando únicamente poniendo el nombre y el tipo o únicamente el nombre.
3. La última de las zonas incluye cada una de las funciones que ofrece la clase. De forma parecida a los atributos, sigue el siguiente formato:

   ```
   visibilidad nombre_funcion { parametros } : tipo-devuelto {propiedades}
   ```

   De la misma manera que con los atributos, se suele simplificar indicando únicamente el nombre de la función y, en ocasiones, el tipo devuelto.



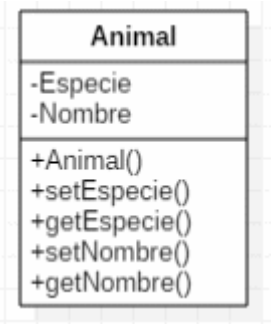
Notación de una clase

Tanto los atributos como las funciones incluyen al principio de su descripción la visibilidad que tendrá. Esta visibilidad se identifica escribiendo un símbolo y podrá ser:

(+) Pública. Representa que se puede acceder al atributo o función desde cualquier lugar de la aplicación.

(-) Privada. Representa que se puede acceder al atributo o función únicamente desde la misma clase.

(#) Protegida. Representa que el atributo o función puede ser accedida únicamente desde la misma clase o desde las clases que hereden de ella (clases derivadas).



Multiplicidad

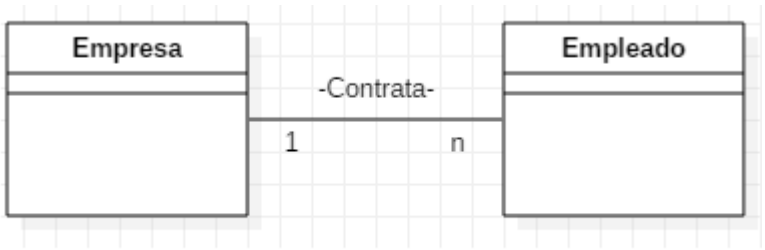
Es decir, el número de elementos de una clase que participan en una relación. Se puede indicar un número, un rango… Se utiliza n o * para identificar un número cualquiera.

Tipos de multiplicidad

- Uno a uno 1 - 1
- Uno a muchos 1 - 1..*
- Uno a muchos o ninguno 1 – 0..*

Nombre de la relación.

En ocasiones se escriba una indicación de la asociación que ayuda a entender la relación que tienen dos clases. Suelen utilizarse verbos como, por ejemplo: “Una empresa contrata a n empleados”



Ejemplo de relación Empresa-Empleado

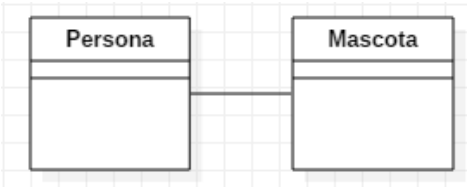
Tipos de relaciones

- Asociación.
- Agregación.
- Composición.
- Generalización (Herencia)

Asociación

Este tipo de relación es el más común y se utiliza para representar dependencia semántica. Se representa con una simple línea continua que une las clases que están incluidas en la asociación.

Un ejemplo de asociación podría ser: “Una mascota pertenece a una persona”.

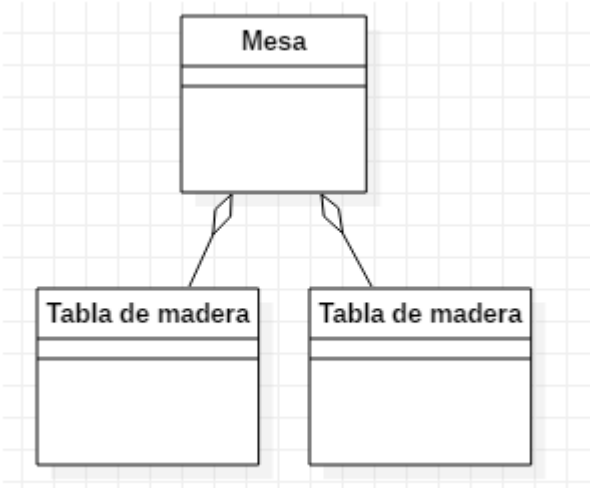


Agregación

Es una representación jerárquica que indica a un objeto y las partes que componen ese objeto. Es decir, representa relaciones en las que un objeto es parte de otro, pero aun así debe tener existencia en sí mismo.

Se representa con una línea que tiene un rombo en la parte de la clase que es una agregación de la otra clase (es decir, en la clase que contiene las otras).

Un ejemplo de esta relación podría ser: “Las mesas están formadas por tablas de madera y tornillos o, dicho de otra manera, los tornillos y las tablas forman parte de una mesa”. Como ves, el tornillo podría formar parte de más objetos, por lo que interesa especialmente su abstracción en otra clase.



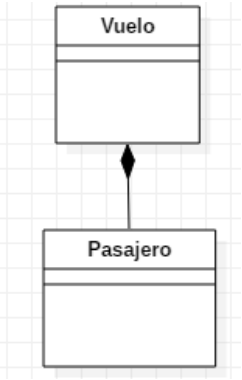
Ejemplo de agregación

Composición

La composición es similar a la agregación, representa una relación jerárquica entre un objeto y las partes que lo componen, pero de una forma más fuerte. En este caso, los elementos que forman parte no tienen sentido de existencia cuando el primero no existe. Es decir, cuando el elemento que contiene los otros desaparece, deben desaparecer todos ya que no tienen sentido por sí mismos, sino que dependen del elemento que componen. Además, suelen tener los mismos tiempos de vida. Los componentes no se comparten entre varios elementos, esta es otra de las diferencias con la agregación.

Se representa con una línea continua con un rombo relleno en la clase que es compuesta.

Un ejemplo de esta relación sería: “Un vuelo de una compañía aérea está compuesto por pasajeros, que es lo mismo que decir que un pasajero está asignado a un vuelo”



Diferencia entre agregación y composición

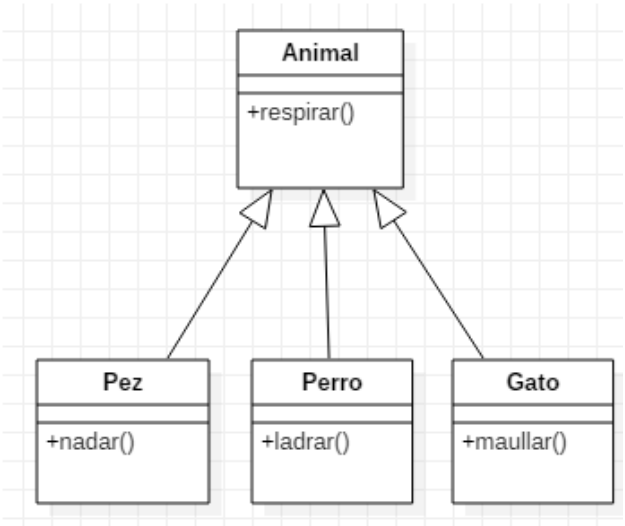
Un “agregado” representa un todo que comprende varias partes; de esta manera, un Comité es un agregado de sus Miembros. Una reunión es un agregado de una agenda, una sala y los asistentes. En el momento de la implementación, esta relación no es de contención. (Una reunión no contiene una sala).

La composición, por otro lado, implica un acoplamiento aún más estricto que la agregación, y definitivamente implica la contención. El requisito básico es que, si una clase de objetos (llamado “contenedor”) se compone de otros objetos (llamados “elementos”), entonces los elementos aparecerán y también serán destruidos como un efecto secundario de crear o destruir el contenedor.

Generalización

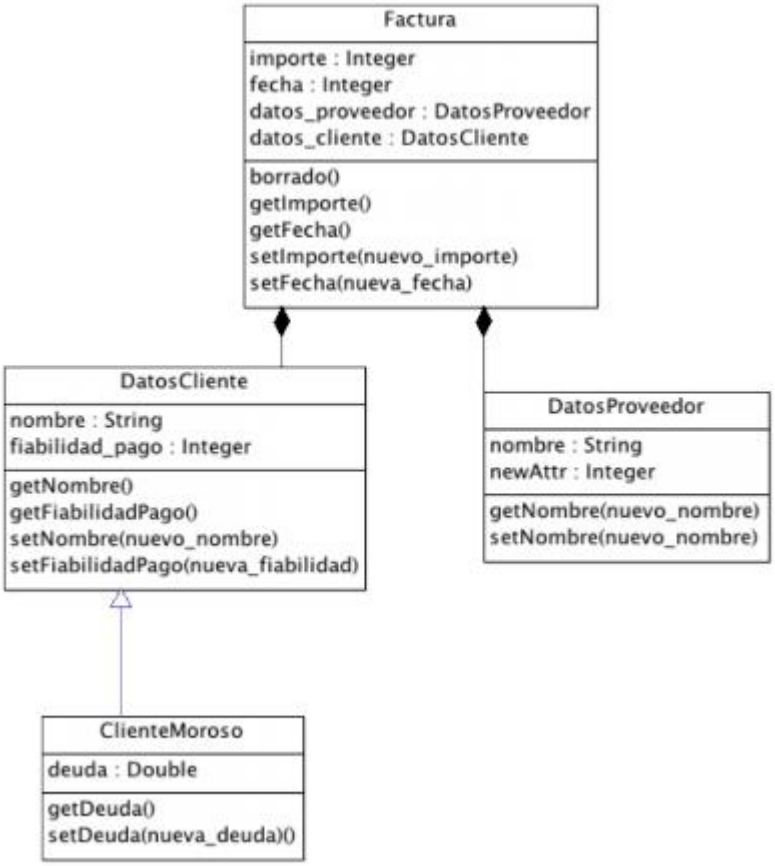
Otra relación muy común en el diagrama de clases es la herencia. Este tipo de relaciones permiten que una clase (clase hija o subclase) reciba los atributos y métodos de otra clase (clase padre o superclase). Estos atributos y métodos recibidos se suman a los que la clase tiene por sí misma. Se utiliza en relaciones “es un”. Un ejemplo de esta relación podría ser la siguiente: Un pez, un perro y un gato son animales.

En este ejemplo, las tres clases (Pez, Perro, Gato) podrán utilizar la función respirar, ya que lo heredan de la clase animal, pero solamente la clase Pez podrá nadar, la clase Perro ladrar y la clase Gato maullar. La clase Animal podría plantearse ser definida abstracta, aunque no es necesario.

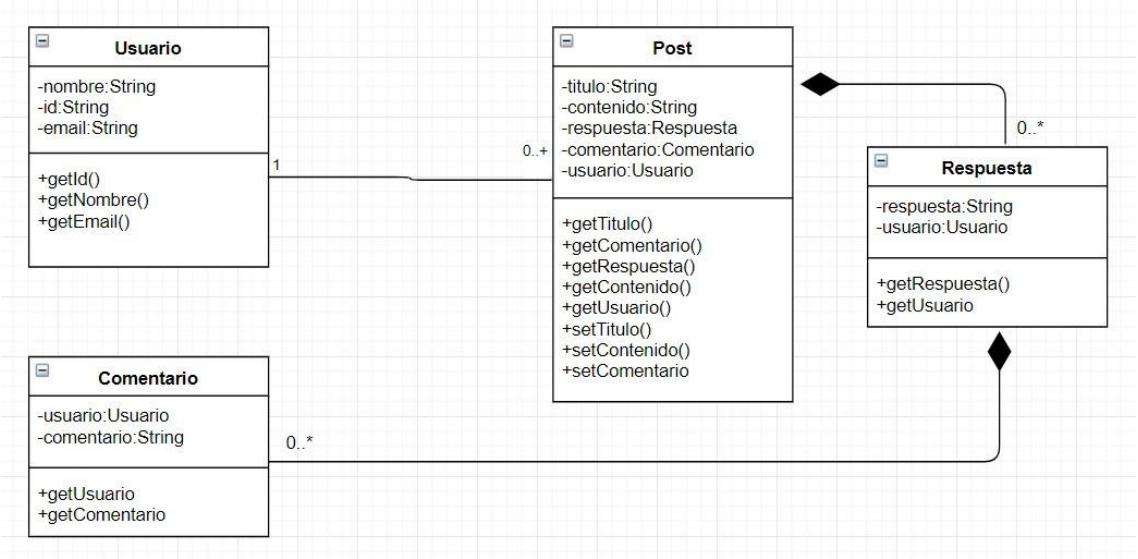


Ejemplo de herencia

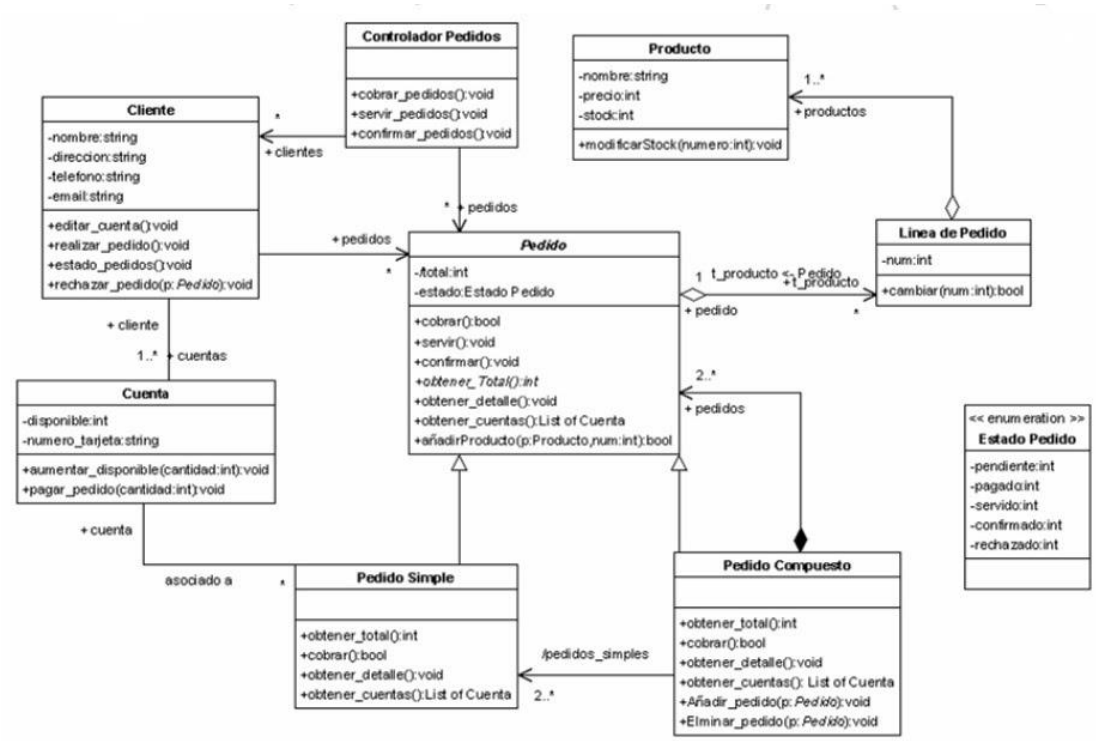
Ejemplo 1. Diagrama de clases de los datos de clientes y personas en una factura.



Ejemplo 2. Diagrama de clases de publicaciones (post) en redes sociales.



Ejemplo 3. Diagrama de clases de pedidos de productos en línea.



4.6. Diseño arquitectónico del software

El diseño arquitectónico se interesa por entender cómo debe organizarse un sistema y cómo tiene que diseñarse la estructura global de ese sistema. En el modelo del proceso de desarrollo de software, como se mostró en el capítulo 2, el diseño arquitectónico es la primera etapa en el proceso de diseño del software. Es el enlace crucial entre el diseño y la ingeniería de requerimientos, ya que identifica los principales componentes estructurales en un sistema y la relación entre ellos. La salida del proceso de diseño arquitectónico consiste en un modelo arquitectónico que describe la forma en que se organiza el sistema como un conjunto de componentes en comunicación.

En los procesos ágiles, por lo general se acepta que una de las primeras etapas en el proceso de desarrollo debe preocuparse por establecer una arquitectura global del sistema. Usualmente no resulta exitoso el desarrollo incremental de arquitecturas. Mientras que la refactorización de componentes en respuesta a los cambios suele ser relativamente fácil, tal vez resulte costoso refactorizar una arquitectura de sistema.

La arquitectura de software es importante porque afecta el desempeño y la potencia, así como la capacidad de distribución y mantenimiento de un sistema (Bosch, 2000). Como afirma Bosch, los componentes individuales implementan los requerimientos funcionales del sistema. Los requerimientos no funcionales dependen de la arquitectura del sistema, es decir, la forma en que dichos componentes se organizan y comunican. En muchos sistemas, los requerimientos no funcionales están también influidos por componentes individuales, pero no hay duda de que la arquitectura del sistema es la influencia dominante. Bass y sus colaboradores (2003) analizan tres ventajas de diseñar y documentar de manera explícita la arquitectura de software:

1. **Comunicación con los participantes** La arquitectura es una presentación de alto nivel del sistema, que puede usarse como un enfoque para la discusión de un amplio número de participantes.

2. **Análisis del sistema** En una etapa temprana en el desarrollo del sistema, aclarar la arquitectura del sistema requiere cierto análisis. Las decisiones

de diseño arquitectónico tienen un efecto profundo sobre si el sistema puede o no cubrir requerimientos críticos como rendimiento, fiabilidad y mantenibilidad.

3. **Reutilización** a gran escala Un modelo de una arquitectura de sistema es una descripción corta y manejable de cómo se organiza un sistema y cómo interoperan sus componentes. Por lo general, la arquitectura del sistema es la misma para sistemas con requerimientos similares y, por lo tanto, puede soportar reutilización de software a gran escala. Es posible desarrollar arquitecturas de línea de productos donde la misma arquitectura se reutilice mediante una amplia gama de sistemas relacionados.

Debido a la estrecha relación entre los requerimientos no funcionales y la arquitectura de software, el estilo y la estructura arquitectónicos particulares que se elijan para un sistema dependerán de los requerimientos de sistema no funcionales:

1. **Rendimiento** Si el rendimiento es un requerimiento crítico, la arquitectura debe diseñarse para localizar operaciones críticas dentro de un pequeño número de componentes, con todos estos componentes desplegados en la misma computadora en vez de distribuirlos por la red. Esto significaría usar algunos componentes relativamente grandes, en vez de pequeños componentes de grano fino, lo cual reduce el número de comunicaciones entre componentes. También puede considerar organizaciones del sistema en tiempo de operación que permitan a este ser replicable y ejecutable en diferentes procesadores.

2. **Seguridad** Si la seguridad es un requerimiento crítico, será necesario usar una estructura en capas para la arquitectura, con los activos más críticos protegidos en las capas más internas, y con un alto nivel de validación de seguridad aplicado a dichas capas.

3. **Protección** Si la protección es un requerimiento crítico, la arquitectura debe diseñarse de modo que las operaciones relacionadas con la protección se ubiquen en algún componente individual o en un pequeño

número de componentes. Esto reduce los costos y problemas de validación de la protección, y hace posible ofrecer sistemas de protección relacionados que, en caso de falla, desactiven con seguridad el sistema.

4. **Disponibilidad** Si la disponibilidad es un requerimiento crítico, la arquitectura tiene que diseñarse para incluir componentes redundantes de manera que sea posible sustituir y actualizar componentes sin detener el sistema. En el capítulo 13 se describen dos arquitecturas de sistema tolerantes a fallas en sistemas de alta disponibilidad.

5. **Mantenibilidad** Si la mantenibilidad es un requerimiento crítico, la arquitectura del sistema debe diseñarse usando componentes autocontenidos de grano fino que puedan cambiarse con facilidad. Los productores de datos tienen que separarse de los consumidores y hay que evitar compartir las estructuras de datos.

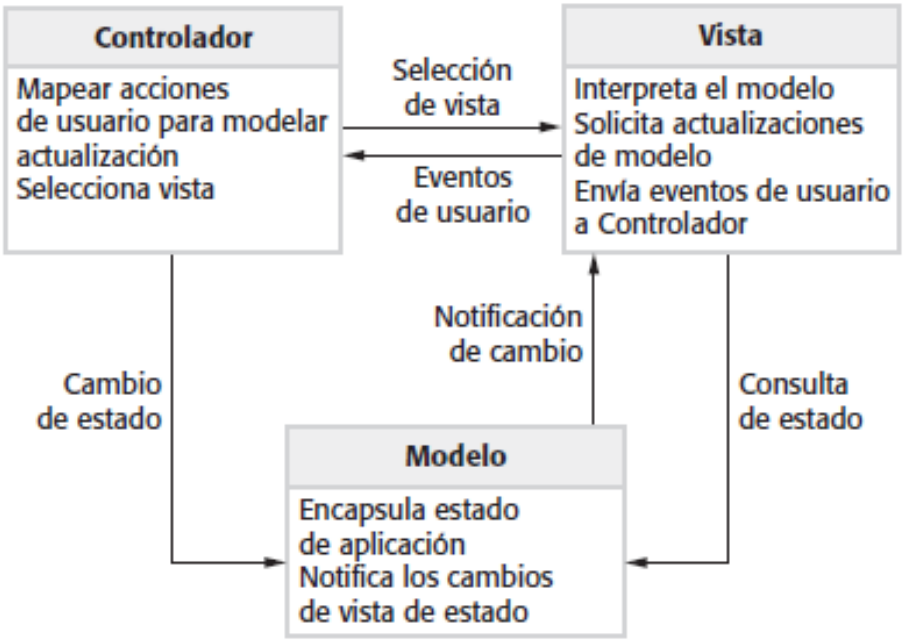
4.6.1. Patrón arquitectónico

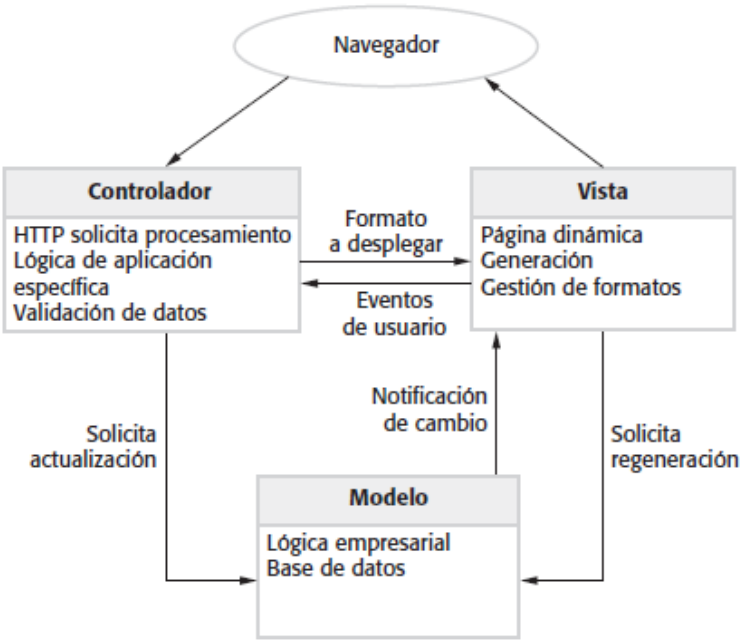
Un patrón arquitectónico se puede considerar como una descripción abstracta estilizada de buena práctica, que se ensayó y puso a prueba en diferentes sistemas y entornos. De este modo, un patrón arquitectónico debe describir una organización de sistema que ha tenido éxito en sistemas previos. Debe incluir información sobre cuándo es y cuándo no es adecuado usar dicho patrón, así como sobre las fortalezas y debilidades del patrón. Por ejemplo, el muy conocido patrón Modelo-Vista-Controlador.

Este patrón es el soporte del manejo de la interacción en muchos sistemas basados en la Web. La descripción del patrón estilizado incluye el nombre del patrón, una breve descripción (con un modelo gráfico asociado) y un ejemplo del tipo de sistema donde se usa el patrón (de nuevo, quizá con un modelo gráfico).

4.6.2. El patrón modelo vista controlador

Nombre	MVC (modelo de vista del controlador)
Descripción	Separa presentación e interacción de los datos del sistema. El sistema se estructura en tres componentes lógicos que interactúan entre sí. El componente Modelo maneja los datos del sistema y las operaciones asociadas a esos datos. El componente Vista define y gestiona cómo se presentan los datos al usuario. El componente Controlador dirige la interacción del usuario (por ejemplo, teclas oprimidas, clics del mouse, etcétera) y pasa estas interacciones a Vista y Modelo. Véase la figura 6.3.
Ejemplo	La figura 6.4 muestra la arquitectura de un sistema de aplicación basado en la Web, que se organiza con el uso del patrón MVC.
Cuándo se usa	Se usa cuando existen múltiples formas de ver e interactuar con los datos. También se utiliza al desconocerse los requerimientos futuros para la interacción y presentación.
Ventajas	Permite que los datos cambien de manera independiente de su representación y viceversa. Soporta en diferentes formas la presentación de los mismos datos, y los cambios en una representación se muestran en todos ellos.
Desventajas	Puede implicar código adicional y complejidad de código cuando el modelo de datos y las interacciones son simples.

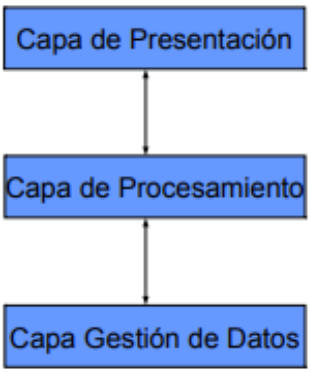




4.6.3. Patrón cliente-servidor

El diseño de sistemas cliente-servidor debería reflejar la estructura lógica de la aplicación.

Una forma de mirar a una aplicación es la siguiente:



Capa de presentación

Presentación de información al usuario e interacción con el mismo

Capa de procesamiento

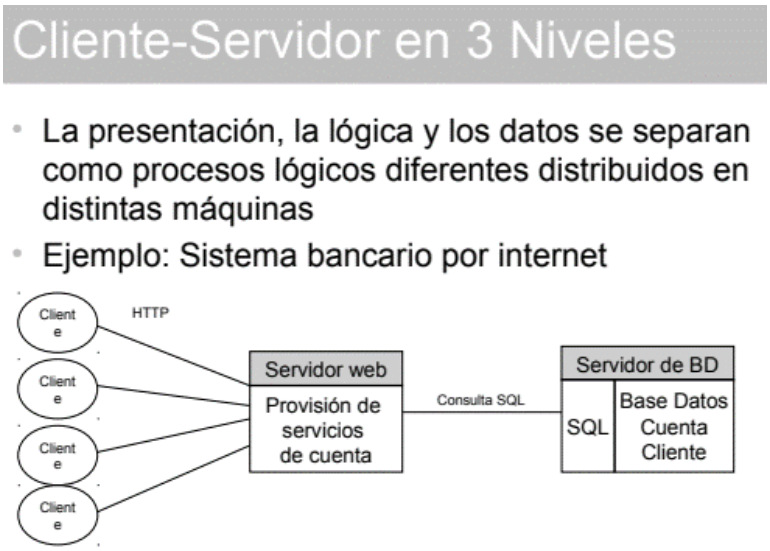
Implementación de la lógica de la aplicación

Capa gestión de datos

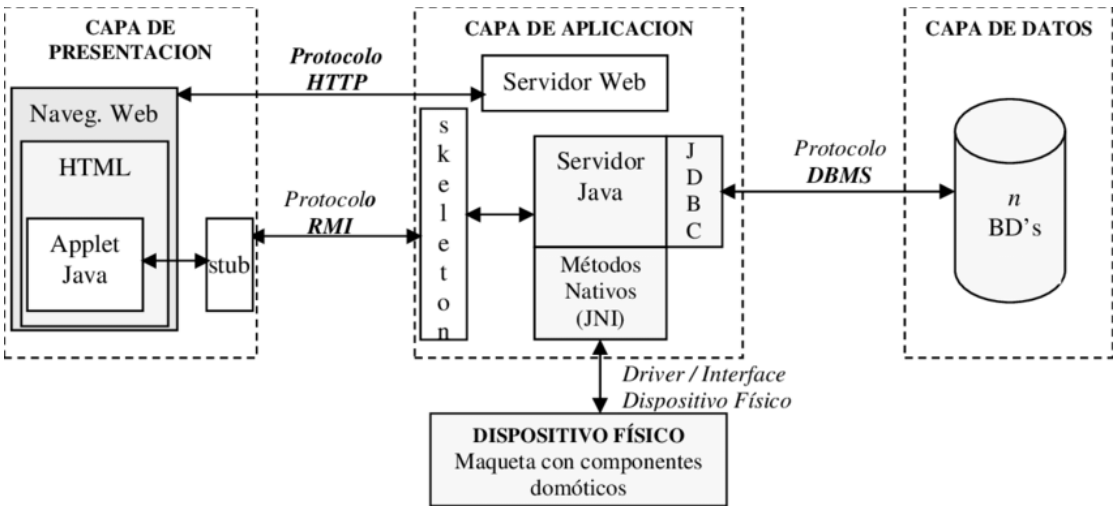
Operaciones de bases de datos y archivos

Ejemplos de diseños de arquitectura de software

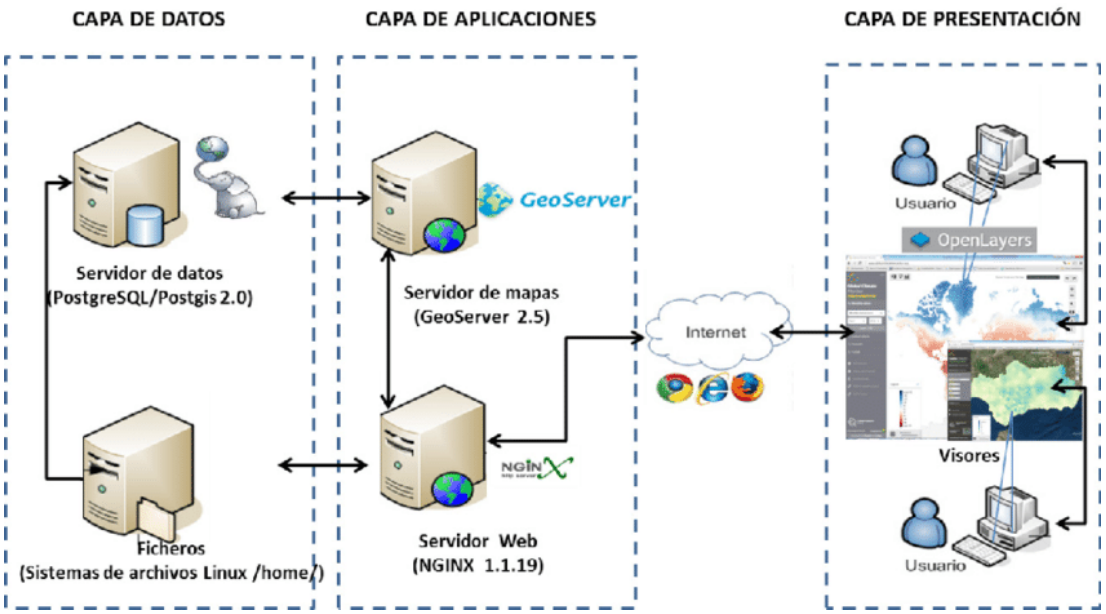
Ejemplo 1



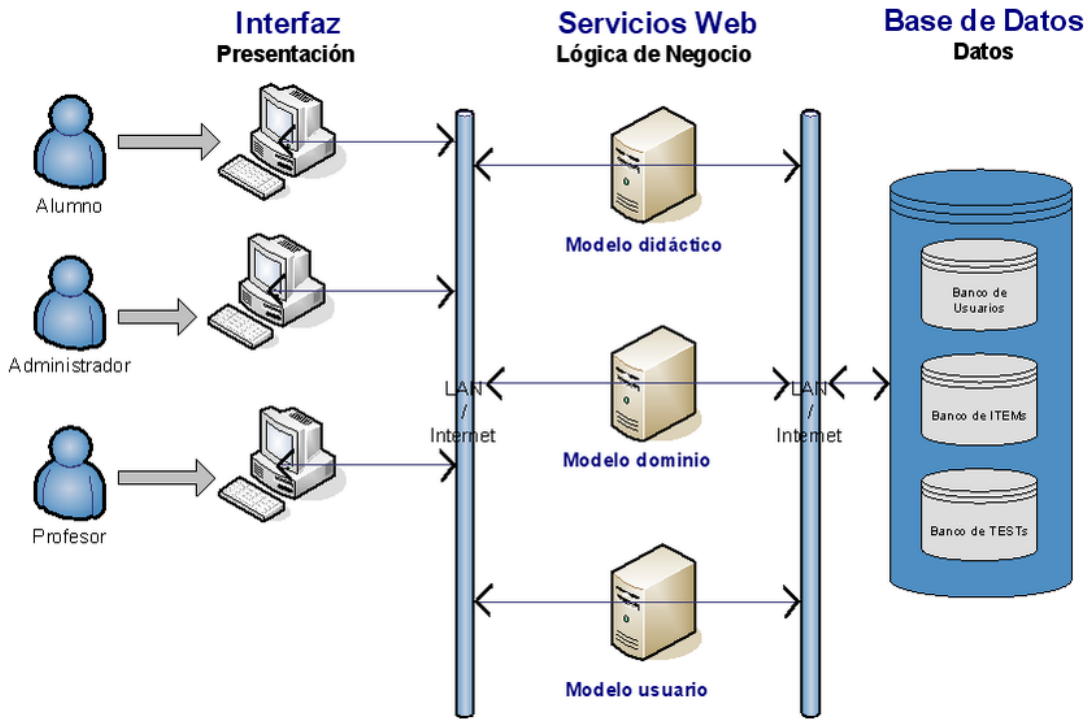
Ejemplo 2



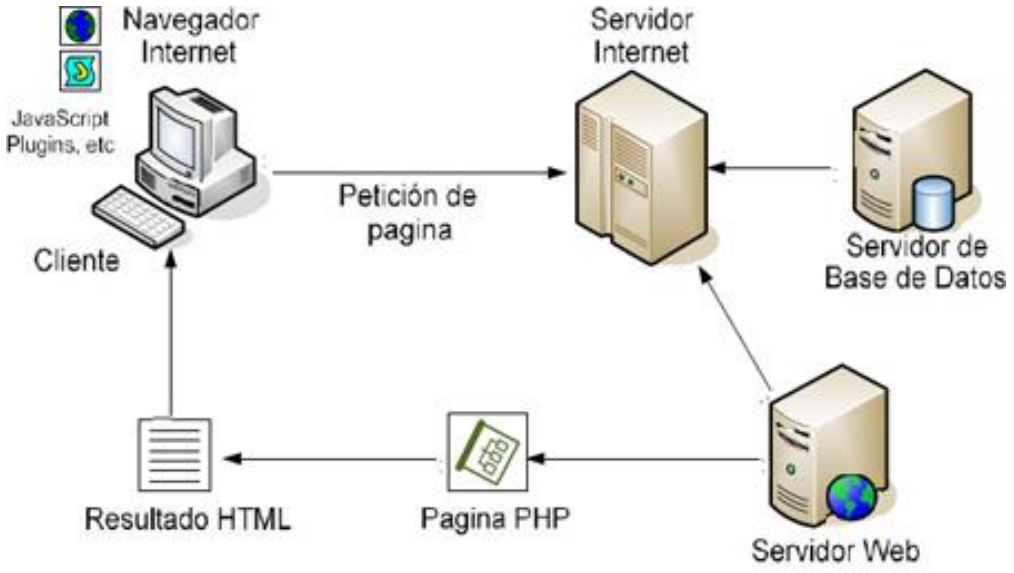
Ejemplo 3



Ejemplo 4



Ejemplo 5



Programación en el Cliente y en el Servidor

Referencias Bibliográficas

Referencias Bibliográficas

Bass, L., Clements, P. y Kazman, R. (2003). Software Architecture in Practice, 2a ed. Boston: Addison-Wesley.

Batista, M (2020). *Origen y Evolución*. Software. Obtenido de: http://softwera-unesr.blogspot.com/p/origen-y-evolucion_23.html

Beck, K. (1999). “Embracing Change with Extreme Programming”. IEEE Computer, 32 (10), 70–8.

Bosch, J. (2000). Design and Use of Software Architectures. Harlow, UK: Addison-Wesley

Cabot, S., J. (2013) Ingeniería del software, Editorial UOC. ProQuest Ebook Central: https://ebookcentral.proquest.com/lib/uteclvtsp/detail.action?docID=3219169

Campderrich, F. (2003) Ingeniería del software, Editorial UOC, ProQuest Ebook Central: https://ebookcentral.proquest.com/lib/uteclvtsp/detail.action?docID=3206903

Davis, A. M. (1993). Software Requirements: Objects, Functions and States. Englewood Cliffs, NJ: Prentice Hall.

Dijkstra, E. W., Dahl, O. J. y Hoare, C. A. R. (1972). Structured Programming. Londres: Academic Press.

IEEE. (1998). “IEEE Recommended Practice for Software Requirements Specifications”. En IEEE Software Engineering Standards Collection. Los Alamitos, Ca.: IEEE Computer Society Press.

Jacobson, I., Christerson, M., Jonsson, P. y Overgaard, G. (1993). Object-Oriented Software Engineering. Wokingham: Addison-Wesley.

Kotonya, G. y Sommerville, I. (1998). Requirements Engineering: Processes and Techniques. Chichester, UK: John Wiley and Sons.

Pressman, R. S. (2010). *Ingeniería del software. Un enfoque práctico* (Séptima ed.). México D. F.: The McGraw-Hill.

Rumbaugh, J., Jacobson, I. y Booch, G. (1999). The Unified Software Development Process. Reading, Mass.: Addison-Wesley.

Sommerville, I. (2011). *Ingeniería De Software* (Novena ed.). México: Pearson Educación.

Resumen

En este libro se investigó los fundamentos de la ingeniería del software y se documentó los conceptos básicos del software y los procesos para la creación de software, se documentó también cómo llevar a cabo un análisis de requerimientos de software, cuáles son los tipos de requerimientos, cómo obtenerlos, documentarlos, validarlos y modelarlos mediante diferentes tipos de diagramas. Este libro tiene como objetivo aplicar los conceptos y teorías de administración para la creación de software de calidad mediante las diferentes metodologías, técnicas y herramientas de la ingeniería del software para que el profesional de sistemas informáticos sea capaz de gestionar con eficiencia y en ambientes éticos y de responsabilidad proyectos inmersos en la creación software. Se investigó varios libros llegando a realizar un compendio de las partes más importantes y se llegó a la conclusión de que la ingeniería del software proporciona varias metodologías para el desarrollo del software y varios modelos para el diseño de software que el profesional puede utilizar dependiendo las diferentes necesidades o las diferentes situaciones o proyectos a desarrollar.

Editorial Grupo AEA

www.grupo-aea.com

www.editorialgrupo-aea.com

Grupo de Asesoría Empresarial & Académica

Grupoaea.ecuador

Editorial Grupo AEA

ISBN: 978-9942-7014-7-3