Procesos de Software

Un proceso de software es una serie de actividades necesarias para desarrollar un producto de software. Estas actividades pueden abarcar desde el desarrollo desde cero utilizando lenguajes de programación estándar hasta la configuración e integración de software comercial o componentes del sistema. Los procesos de software deben incluir las siguientes actividades fundamentales:

1. **Especificación del software:** Definir la funcionalidad y restricciones del software.
2. **Diseño e implementación:** Desarrollar el software de acuerdo con las especificaciones.
3. **Validación del software:** Asegurarse de que el software cumple con las expectativas del cliente.
4. **Evolución del software:** Adaptar el software a las necesidades cambiantes del cliente.

Además, los procesos incluyen actividades complejas como validación de requisitos, diseño arquitectónico, pruebas de unidad, documentación y manejo de la configuración del software.

Componentes Clave de los Procesos de Software

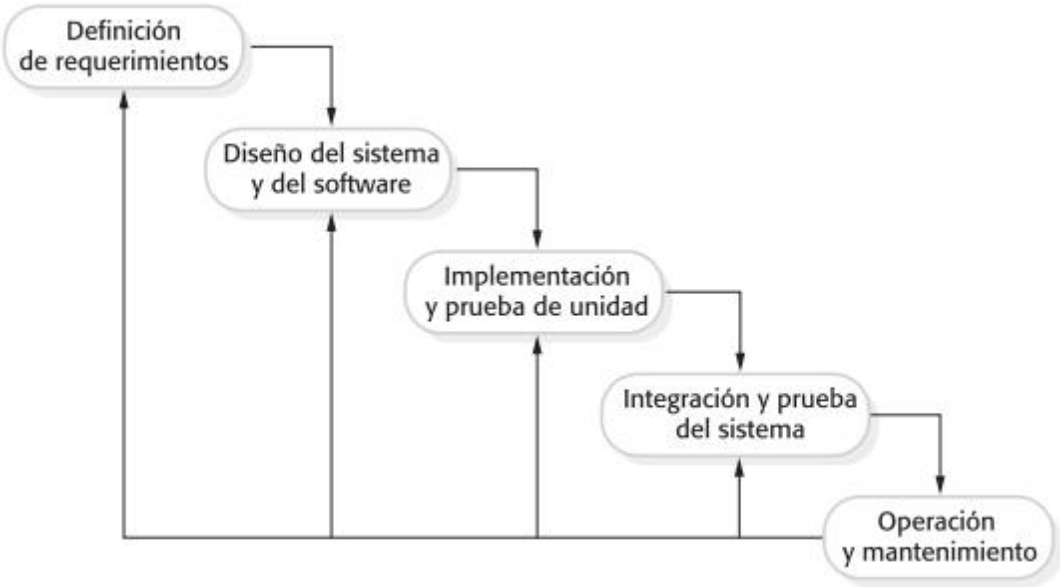
- **Productos:** Resultados tangibles de una actividad del proceso (ej., modelo de arquitectura de software).
- **Roles:** Responsabilidades de las personas en el proceso (ej., gerente de proyecto, programador).
- **Precondiciones y postcondiciones:** Reglas antes y después de cada actividad del proceso.

Tipos de Procesos de Software

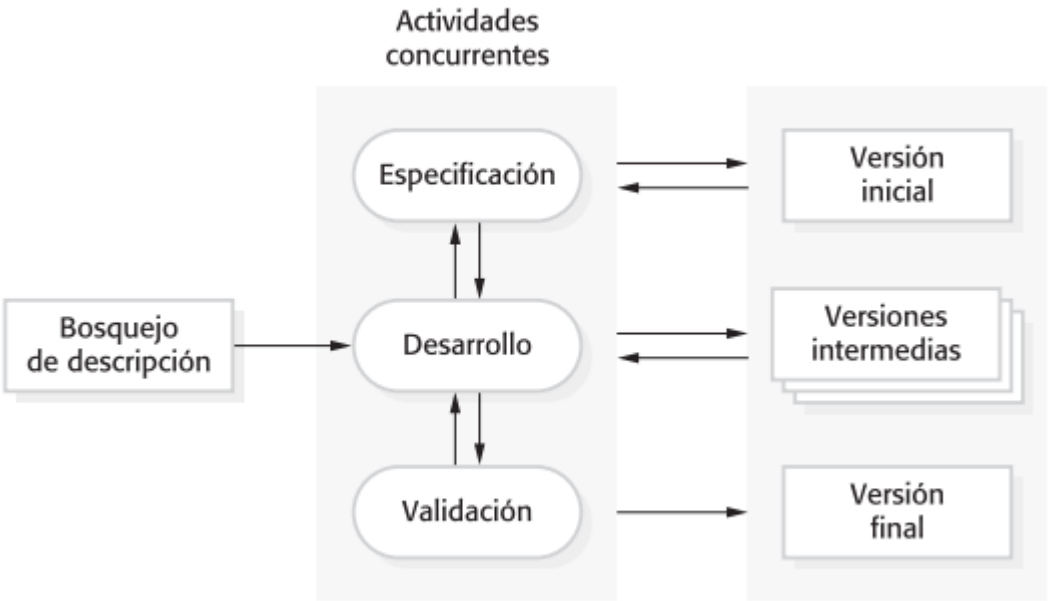
1. **Procesos dirigidos por un plan (plan-driven):** Todas las actividades son planificadas previamente.
2. **Procesos ágiles:** Enfocados en la adaptabilidad y cambios rápidos según los requisitos del cliente.

Modelos de Proceso de Software

1. **Modelo en cascada (waterfall):** Secuencial, con fases de especificación, desarrollo, validación y evolución.



2. **Desarrollo incremental:** Desarrollo en versiones pequeñas e iterativas con retroalimentación continua del cliente.



3. **Ingeniería de software orientada a la reutilización:** Se basa en la integración de componentes existentes en lugar de desarrollar desde cero.

Figura 2.3 Ingeniería de software orientada a la reutilización

Ventajas del Desarrollo Incremental frente al Modelo en Cascada

- Mayor adaptabilidad a cambios en los requisitos del cliente.
- Retroalimentación continua y temprana del cliente.
- Más rápida entrega de versiones del software.

Consideraciones para Elegir un Modelo de Software

El modelo en cascada es adecuado cuando los requisitos están bien definidos y no se esperan cambios importantes. Por otro lado, el desarrollo incremental y los procesos ágiles son ideales para proyectos con alta incertidumbre o con necesidad de adaptabilidad continua.

Desarrollo Incremental

El desarrollo incremental se ha convertido en el enfoque más común para desarrollar sistemas de aplicación. Este enfoque puede combinarse con un enfoque basado en un plan o con metodologías ágiles, permitiendo identificar prioridades tempranas y adaptarse al avance del proyecto y a las necesidades del cliente.

Desafíos del Desarrollo Incremental

1. **Falta de visibilidad del proceso:** Los administradores necesitan entregas regulares para medir el avance, pero el desarrollo rápido puede dificultar la documentación precisa.
2. **Degradación de la estructura del sistema:** Los cambios frecuentes pueden corromper la arquitectura del software, volviendo el mantenimiento más complejo y costoso.

El enfoque incremental es ideal para proyectos empresariales y permite realizar entregas tempranas para obtener retroalimentación continua del cliente. Sin embargo, puede no ser adecuado para sistemas muy grandes o complejos sin una arquitectura bien definida.

Ingeniería de Software Orientada a la Reutilización

En este enfoque se prioriza la integración de componentes de software existentes (COTS - Commercial Off-The-Shelf) para acelerar el desarrollo y reducir costos y riesgos. Se enfoca en:

1. **Análisis de componentes:** Búsqueda de componentes existentes que cumplan con los requisitos.
2. **Modificación de requisitos:** Adaptar los requisitos a las capacidades de los componentes reutilizables.
3. **Diseño del sistema con reutilización:** Integrar componentes reutilizables en el diseño del sistema.
4. **Desarrollo e integración:** Implementar y probar el sistema completo.

Tipos de Componentes Reutilizables

1. **Servicios Web:** Integración de servicios estándares para uso remoto.
2. **Colecciones de objetos:** Uso de frameworks como .NET o J2EE.
3. **Sistemas independientes:** Configuración de sistemas específicos para entornos concretos.

Ventajas y Desafíos

La reutilización permite un desarrollo más rápido y reduce la cantidad de software a desarrollar. No obstante, puede implicar pérdida de control sobre la evolución del sistema y depender de componentes externos.

Actividades del Proceso de Software

Las actividades esenciales incluyen:

- **Especificación:** Definir lo que se espera del software.
- **Diseño:** Convertir especificaciones en un sistema operativo.
- **Implementación:** Programar y probar el software.
- **Evolución:** Adaptar el software a nuevos requisitos.

Especificación del Software

Esta etapa se enfoca en cuatro actividades principales en el proceso de ingeniería de requerimientos:

1. **Estudio de factibilidad:** Evaluar si el proyecto es viable.
2. **Análisis de requisitos:** Obtener requisitos mediante observación y consulta con usuarios.
3. **Especificación de requisitos:** Documentar tanto los requisitos del usuario como del sistema.
4. **Validación de requisitos:** Asegurar que los requisitos sean completos y realistas.

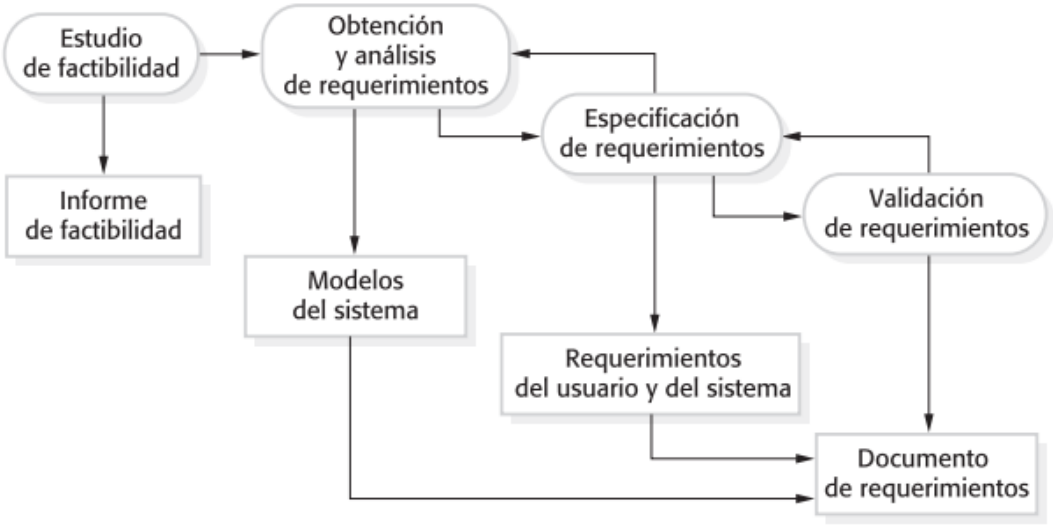


Figura 2.4 Proceso de ingeniería de requerimientos

Diseño e Implementación del Software

La implementación convierte las especificaciones en un sistema operativo. Implica:

- **Diseño arquitectónico:** Identificar componentes principales y su relación.
- **Diseño de interfaz:** Definir cómo interactúan los componentes.
- **Diseño de componentes:** Especificar la funcionalidad de cada módulo.
- **Diseño de base de datos:** Estructurar cómo se manejarán los datos.

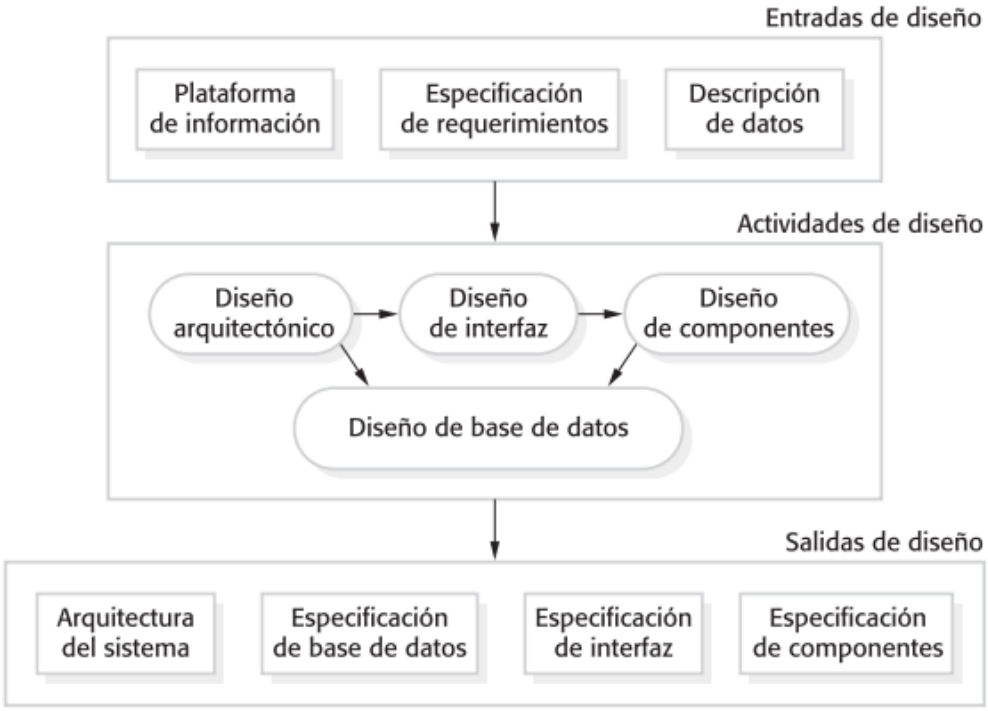


Figura 2.5 Modelo general del proceso de diseño

Validación del Software

La validación verifica que el software cumpla con los requisitos y las expectativas del cliente. Incluye:

1. **Pruebas de componentes:** Evaluar módulos individuales.
2. **Pruebas del sistema:** Comprobar la integración de todos los componentes.
3. **Pruebas de aceptación:** Validar que el sistema cumpla con los requisitos del cliente en un entorno real.

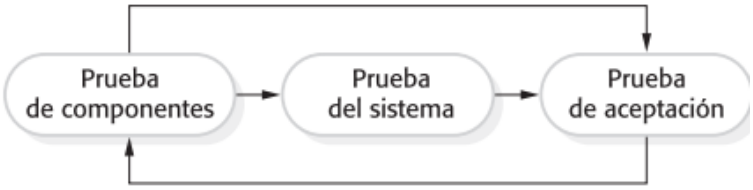


Figura 2.6 Etapas de pruebas

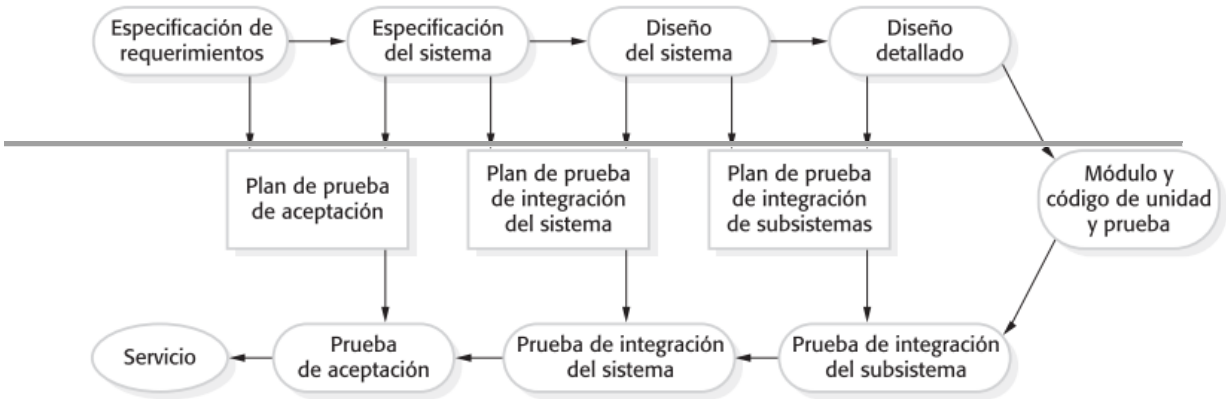
Procesos de Pruebas en el Desarrollo de Software

Pruebas de Componentes y Proceso Incremental

- En el enfoque incremental, cada incremento del software se pone a prueba a medida que se diseña, basándose en los requisitos de cada iteración.
- En programación extrema, las pruebas se desarrollan junto con los requisitos, asegurando que no haya demoras en la creación de casos de prueba.
- En procesos dirigidos por un plan, se utilizan conjuntos de planes de prueba preformulados, conocidos como "modelo V de desarrollo".

Pruebas Alfa y Beta

- **Pruebas Alfa:** Se realizan internamente con un cliente limitado, permitiendo ajustes antes de la implementación masiva.
- **Pruebas Beta:** Se entregan versiones preliminares del software a un grupo de usuarios finales para obtener retroalimentación real antes del lanzamiento oficial.



Evolución del Software

Flexibilidad y Mantenimiento

- La flexibilidad del software permite cambios continuos a lo largo de su vida útil, lo que es más barato y práctico que realizar cambios en el hardware.
- Históricamente, se separaba el desarrollo del mantenimiento, pero ahora se considera un proceso continuo y evolutivo.

Enfrentando el Cambio

- Los cambios en los requisitos pueden generar costos adicionales y la necesidad de rehacer partes del sistema.
- **Enfoques para manejar el cambio:**
    1. **Evitar el cambio:** Anticipar posibles ajustes mediante el uso de prototipos y pruebas tempranas.
    2. **Tolerancia al cambio:** Diseñar el proceso de manera que los cambios sean menos costosos, aplicando metodologías como el desarrollo incremental.

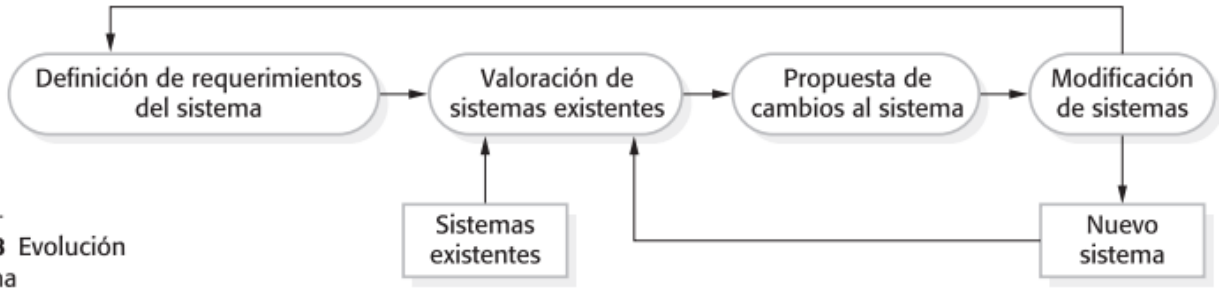


Figura 2.8 Evolución del sistema

Prototipos en el Desarrollo de Software

Ventajas del Uso de Prototipos

- Permiten experimentar con el sistema antes de su implementación final.
- Ayudan en la selección y validación de requisitos y en el diseño de interfaces de usuario.

Limitaciones del Prototipo

- No siempre cubre requisitos no funcionales (rendimiento, seguridad, fiabilidad).
- El desarrollo rápido puede implicar una falta de documentación adecuada.

- Los cambios en el prototipo pueden afectar negativamente la estructura final del sistema.

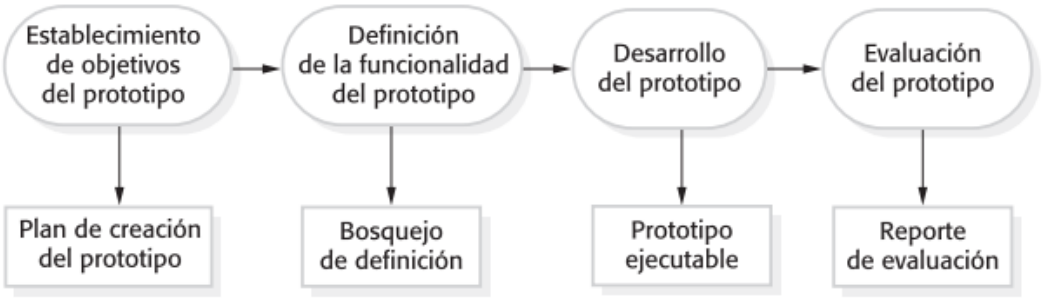


Figura 2.9 Proceso de desarrollo del prototipo

Entrega Incremental

Características

- Entrega partes funcionales del software en incrementos controlados.
- Permite a los clientes utilizar partes del sistema antes de que esté completamente desarrollado.

Ventajas

1. Los clientes pueden probar versiones preliminares y dar retroalimentación.
2. Se pueden priorizar características críticas del sistema.
3. El proceso es más adaptable a cambios de requisitos.

Desafíos

- Dificultad para identificar recursos comunes necesarios en todos los incrementos.
- Puede ser complicado reemplazar sistemas antiguos con nuevos sistemas iterativos.
- La especificación completa del sistema puede ser incierta hasta etapas avanzadas.

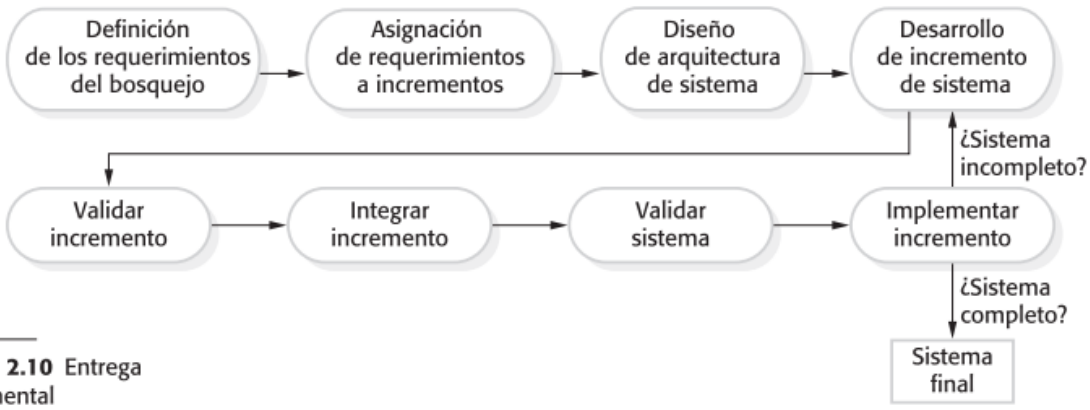


Figura 2.10 Entrega incremental

Modelo en Espiral de Boehm

Enfoque Dirigido por el Riesgo

- Combina la evitación del cambio con la tolerancia al cambio.
- Se estructura en ciclos que permiten evaluar riesgos y planificar estrategias para mitigarlos.

Fases del Ciclo en Espiral

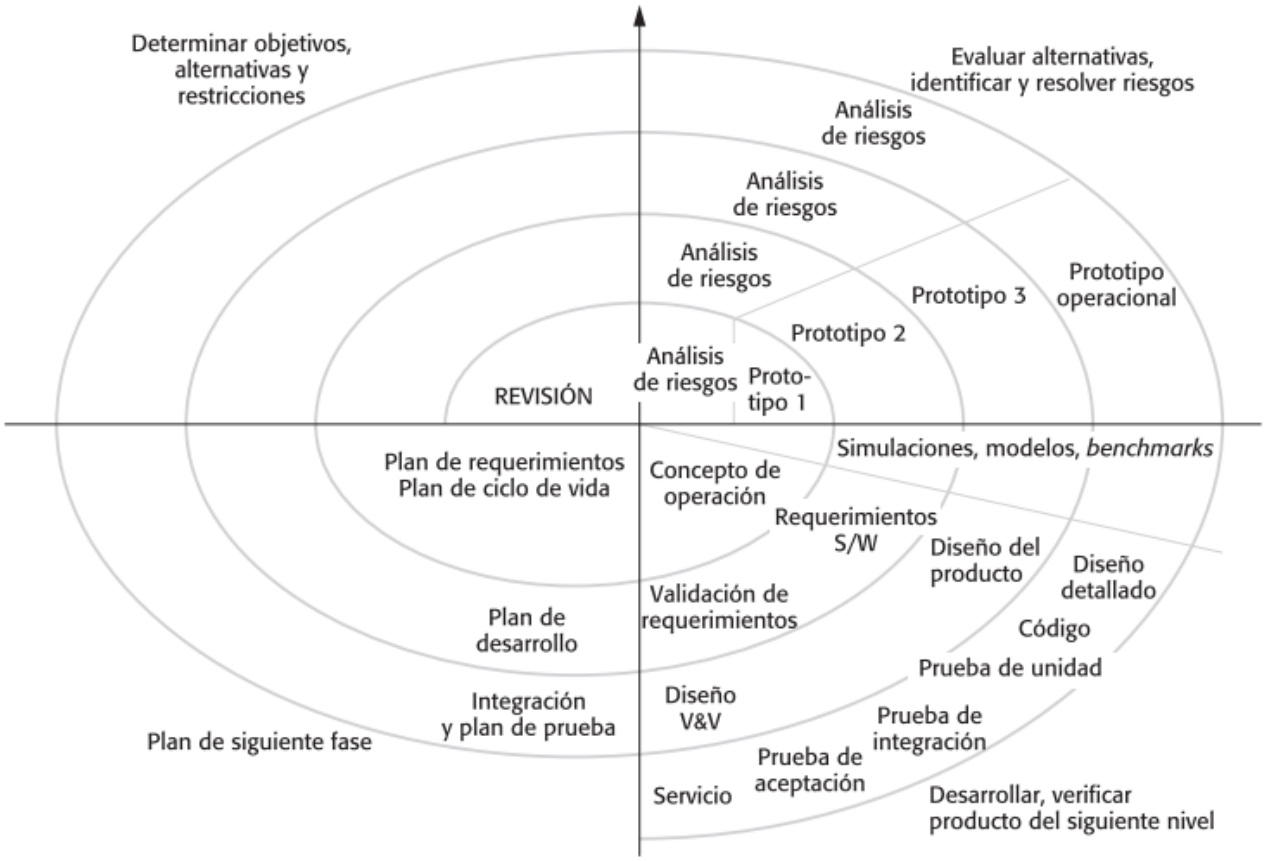
1. **Establecimiento de objetivos:** Definir metas específicas para cada fase del proyecto.
2. **Valoración y reducción del riesgo:** Evaluar riesgos y tomar acciones para minimizarlos.
3. **Desarrollo y validación:** Elegir el mejor modelo de desarrollo basado en la evaluación de riesgos.
4. **Planeación:** Revisar el proyecto y decidir si continuar con otro ciclo de la espiral.

Características del Modelo

El modelo en espiral se diferencia de otros modelos de proceso de software al reconocer explícitamente el riesgo. Cada ciclo en la espiral comienza con la definición de objetivos de rendimiento y funcionalidad, seguido de la identificación y evaluación de riesgos. Se utiliza un enfoque iterativo para resolver riesgos mediante actividades como análisis detallado, creación de prototipos y simulación.

Enfoque en la Minimización del Riesgo

- El modelo prioriza la reducción de riesgos a través de una planificación proactiva.
- Se evalúan alternativas y se identifican fuentes de riesgo en cada fase del proyecto.
- La gestión de riesgos es un componente esencial de la administración del proyecto.



El Proceso Unificado Racional (RUP)

¿Qué es RUP?

El **Proceso Unificado Racional (RUP)** es un modelo de proceso moderno derivado del trabajo con UML y el desarrollo de software unificado. Es un modelo híbrido que combina aspectos de procesos genéricos como el modelo en cascada, el desarrollo incremental y la reutilización de componentes.

Perspectivas del RUP

1. **Perspectiva dinámica:** Muestra las fases del modelo a lo largo del tiempo.
2. **Perspectiva estática:** Presenta las actividades del proceso establecidas.
3. **Perspectiva práctica:** Sugiere buenas prácticas durante el proceso de desarrollo.

Fases del RUP

1. **Concepción:** Establece un caso empresarial y evalúa la viabilidad del proyecto.
2. **Elaboración:** Desarrolla un marco arquitectónico, define el plan del proyecto y evalúa riesgos clave.
3. **Construcción:** Implementación del software y documentación para la entrega al usuario.
4. **Transición:** Se enfoca en llevar el software al entorno del usuario final, asegurando su correcto funcionamiento.

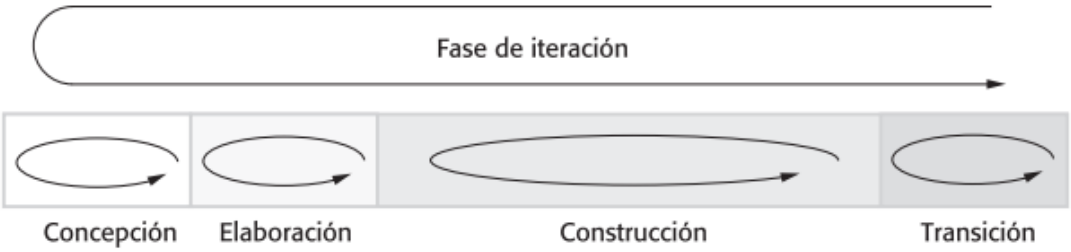


Figura 2.12 Fases en el Proceso Unificado Racional

Flujos de Trabajo en RUP

1. **Modelado del negocio:** Uso de casos de negocio para modelar procesos empresariales.
2. **Requerimientos:** Identificación de actores e interacción con el sistema.
3. **Análisis y diseño:** Creación de modelos arquitectónicos y de componentes.
4. **Implementación:** Estructuración y codificación del sistema.
5. **Pruebas:** Iterativas, integradas con la implementación.
6. **Despliegue:** Liberación del producto y configuración en el entorno del usuario.
7. **Administración de la configuración y del cambio:** Gestión de modificaciones en el sistema.
8. **Administración del proyecto:** Supervisión del desarrollo del software.
9. **Entorno:** Provisión de herramientas y recursos para el equipo de desarrollo.

Buenas Prácticas del RUP

1. **Desarrollo Iterativo:** Priorizar las características más importantes del sistema.
2. **Gestión de Requerimientos:** Documentar explícitamente los requisitos del cliente.
3. **Arquitecturas Basadas en Componentes:** Estructurar el sistema en módulos reutilizables.
4. **Modelado Visual:** Utilizar UML para representar el sistema.
5. **Verificación de Calidad:** Garantizar el cumplimiento de los estándares organizacionales.
6. **Control de Cambios:** Utilizar sistemas y herramientas para gestionar la configuración del software.

Puntos Clave del Desarrollo de Software

- Los procesos de software incluyen actividades para la especificación, diseño, implementación, validación y evolución del sistema.
- Los modelos de proceso generales (cascada, incremental, reutilización) ofrecen diferentes enfoques para organizar el desarrollo de software.
- La ingeniería de requisitos es crucial para alinear las necesidades del cliente con las capacidades del sistema.
- Los procesos de diseño e implementación convierten los requisitos en un sistema funcional.
- La evolución del software permite la adaptación continua a nuevos requisitos.
- Los procesos deben ser adaptables para gestionar el cambio, utilizando prototipos o entregas incrementales.
- El RUP es un modelo moderno que combina diferentes enfoques, proporcionando flexibilidad y buenas prácticas a lo largo del ciclo de vida del software.

Conclusiones

El desarrollo de software requiere una combinación de buenas prácticas en especificación, diseño, implementación y validación. La elección del enfoque (incremental, en cascada o reutilización) dependerá del tipo de proyecto, la complejidad del sistema y las necesidades del cliente.

El desarrollo de software moderno se enfoca en la adaptabilidad y la gestión eficiente de cambios y riesgos. Los enfoques incrementales, el uso de prototipos y

el modelo en espiral permiten una mayor flexibilidad y alineación con las necesidades cambiantes del cliente. Sin embargo, cada método tiene sus propias ventajas y desafíos, y la elección adecuada dependerá del tipo de proyecto y los requisitos específicos.

ANEXOS

Modelo en cascada

El modelo en cascada es un enfoque de desarrollo de software lineal y secuencial, donde cada fase del proyecto debe completarse antes de iniciar la siguiente. Este modelo es especialmente adecuado para ciertos tipos de sistemas que presentan características específicas:

Sistemas con requisitos claramente definidos y estables: Cuando los requisitos del sistema están bien comprendidos y es poco probable que cambien durante el desarrollo, el modelo en cascada proporciona una estructura clara y ordenada.

Ejemplos:

- Sistemas de procesamiento de nóminas.
- Aplicaciones de gestión de inventarios.

Sistemas de pequeño o mediano tamaño: En proyectos más pequeños, donde el alcance es limitado y manejable, el enfoque secuencial del modelo en cascada puede ser eficiente y efectivo.

Ejemplos:

- Aplicaciones de cálculo financiero.
- Sistemas de gestión de bibliotecas.

Sistemas con tecnologías y herramientas bien comprendidas: Si el equipo de desarrollo está familiarizado con las tecnologías y herramientas que se utilizarán, y no se esperan cambios significativos en estas áreas, el modelo en cascada puede ser una elección adecuada.

Ejemplos:

- Sistemas de gestión de bases de datos estándar.
- Aplicaciones de facturación electrónica.

Sistemas con plazos y presupuestos estrictos: La naturaleza estructurada del modelo en cascada facilita la planificación y el seguimiento del progreso, lo que es beneficioso en proyectos con restricciones de tiempo y recursos.

Ejemplos:

- Sistemas de registro de estudiantes para instituciones educativas.
- Aplicaciones de seguimiento de pedidos para pequeñas empresas.

Sistemas donde la calidad y la seguridad son críticas: En sistemas donde la calidad y la seguridad son primordiales, como en aplicaciones médicas o aeroespaciales, el modelo en cascada permite una documentación exhaustiva y una validación rigurosa en cada fase.

Ejemplos:

- Sistemas de control de tráfico aéreo.
- Aplicaciones de monitoreo de pacientes en entornos hospitalarios.

Es importante destacar que, aunque el modelo en cascada ofrece ventajas en escenarios específicos, su rigidez puede ser una limitación en proyectos donde los requisitos son susceptibles de cambiar o evolucionar durante el desarrollo. En tales casos, enfoques más iterativos o ágiles podrían ser más apropiados.

Modelo de desarrollo incremental

El modelo de desarrollo incremental es una metodología en la que el sistema se construye y entrega en pequeñas partes funcionales llamadas incrementos. Cada incremento añade funcionalidad al sistema, permitiendo a los usuarios interactuar con versiones parciales del producto y proporcionando oportunidades para recibir retroalimentación temprana y continua. Este enfoque es especialmente adecuado para ciertos tipos de sistemas que se benefician de una implementación gradual y adaptable. A continuación, se presentan ejemplos de sistemas donde el desarrollo incremental es particularmente efectivo:

Sistemas de comercio electrónico:

Tiendas en línea: Implementar funcionalidades básicas como la navegación de productos y el carrito de compras en las primeras fases, añadiendo posteriormente características como recomendaciones personalizadas, reseñas de clientes y sistemas de pago avanzados.

Aplicaciones móviles:

Aplicaciones de mensajería: Comenzar con funciones esenciales de envío y recepción de mensajes, e incorporar gradualmente características como llamadas de voz y video, integración con redes sociales y opciones de personalización.

Sistemas de gestión empresarial:

Software de planificación de recursos empresariales (ERP): Desarrollar módulos individuales como contabilidad, recursos humanos y gestión de inventario de manera incremental, permitiendo a la organización adaptarse progresivamente al nuevo sistema.

Plataformas educativas en línea:

Sistemas de gestión de aprendizaje (LMS): Iniciar con funcionalidades básicas para la gestión de cursos y usuarios, y añadir paulatinamente herramientas como foros de discusión, evaluaciones en línea y analíticas de aprendizaje.

Aplicaciones financieras:

Sistemas de banca en línea: Implementar servicios básicos de consulta de saldo y transferencias, incorporando posteriormente funcionalidades como pago de facturas, inversiones y asesoramiento financiero personalizado.

Sistemas de atención médica:

Historias clínicas electrónicas: Comenzar con el registro de información básica del paciente y añadir gradualmente módulos para gestión de medicamentos, programación de citas y seguimiento de tratamientos.

Redes sociales:

Plataformas de interacción social: Lanzar con funcionalidades esenciales de creación de perfiles y publicación de contenido, e incorporar con el tiempo características como mensajería privada, eventos y transmisiones en vivo.

Sistemas de control industrial:

Automatización de procesos: Desarrollar inicialmente el control de procesos críticos y añadir de forma incremental funcionalidades de monitoreo, análisis de datos y mantenimiento predictivo.

El desarrollo incremental permite una adaptación continua a las necesidades cambiantes de los usuarios y del mercado, facilitando la incorporación de mejoras basadas en la retroalimentación obtenida durante el proceso de desarrollo. Esta metodología es especialmente útil en entornos dinámicos donde la flexibilidad y la capacidad de respuesta rápida son esenciales para el éxito del proyecto.

Ingeniería de software orientada a la reutilización

La ingeniería de software orientada a la reutilización se centra en la creación de sistemas de software mediante la integración de componentes existentes, lo que permite reducir costos, mejorar la calidad y acelerar el desarrollo. Este enfoque es especialmente adecuado para ciertos tipos de sistemas que pueden beneficiarse de la reutilización de componentes. A continuación, se presentan ejemplos de sistemas donde este modelo es particularmente efectivo:

Sistemas de gestión de contenido (CMS):

Plataformas como Plone: Este CMS está construido sobre el framework Zope, que facilita la reutilización de componentes y la extensión de funcionalidades.

Sistemas de comercio electrónico:

Tiendas en línea: La implementación de plataformas de comercio electrónico puede beneficiarse de la reutilización de módulos existentes, como carritos de compra, pasarelas de pago y sistemas de gestión de inventario.

Aplicaciones empresariales integradas (ERP):

Sistemas modulares: Los ERP suelen estar compuestos por módulos que gestionan diferentes áreas de una empresa, como finanzas, recursos humanos y logística. La reutilización de estos módulos estándar permite personalizar soluciones según las necesidades específicas de cada organización.

Sistemas de gestión de relaciones con clientes (CRM):

Plataformas CRM: Estas aplicaciones pueden construirse reutilizando componentes que gestionan interacciones con clientes, seguimiento de ventas y campañas de marketing, adaptándose a distintos sectores industriales.

Sistemas de automatización industrial:

Controladores y sensores: La reutilización de componentes de software para la gestión de dispositivos industriales permite desarrollar sistemas de control y monitoreo más eficientes y fiables.

Aplicaciones de análisis de datos:

Herramientas de business intelligence: La integración de componentes existentes para la recopilación, procesamiento y visualización de datos facilita la creación de soluciones de análisis adaptadas a las necesidades empresariales.

Sistemas de gestión educativa:

Plataformas de e-learning: La reutilización de módulos para la gestión de cursos, seguimiento de estudiantes y evaluaciones en línea permite desarrollar sistemas educativos personalizados y escalables.

Sistemas de comunicación empresarial:

Plataformas de mensajería y colaboración: La integración de componentes para chat, videoconferencia y gestión de proyectos facilita la creación de entornos colaborativos adaptados a las necesidades de las organizaciones.

La reutilización de componentes en la ingeniería de software no solo optimiza el proceso de desarrollo, sino que también mejora la mantenibilidad y escalabilidad de los sistemas, al basarse en módulos probados y estandarizados.

Cuestionario de Procesamiento de Software

1. ¿Cuál de las siguientes es una actividad fundamental de un proceso de software?

- A) Documentación legal
- B) Validación del software
- C) Diseño gráfico
- D) Redacción publicitaria

2. ¿Qué caracteriza a un proceso dirigido por un plan?

- A) Flexibilidad total
- B) Cambios constantes
- C) Actividades planificadas previamente
- D) Uso exclusivo de software libre

3. ¿Qué ventaja ofrece el desarrollo incremental frente al modelo en cascada?

- A) Menor control de versiones
- B) Menos entregas
- C) Retroalimentación continua del cliente
- D) Menor documentación

4. ¿Cuál de estos componentes es reutilizable en la ingeniería de software?

- A) Manuales de usuario
- B) Servicios web
- C) Diagramas de flujo
- D) Licencias de uso

5. ¿Qué tipo de prueba se realiza en un entorno real con usuarios finales?

- A) Prueba Alfa
- B) Prueba de componente
- C) Prueba del sistema
- D) Prueba Beta

6. ¿Qué fase del RUP incluye la implementación del software?

- A) Concepción
- B) Elaboración
- C) Construcción
- D) Transición

7. ¿Qué se evalúa en el estudio de factibilidad?

- A) Rendimiento del hardware
- B) Documentación requerida
- C) Viabilidad del proyecto
- D) Calidad del equipo de trabajo

8. ¿Qué busca el modelo en espiral minimizar?

- A) Costos de marketing
- B) Riesgos
- C) Tiempo de diseño
- D) Recursos humanos

9. ¿Cuál es una limitación del uso de prototipos?

- A) No permite probar interfaces
- B) No se pueden modificar
- C) Afecta la estructura final del sistema
- D) No es adecuado para sistemas web

10. ¿Qué práctica del RUP implica priorizar funciones del sistema?

- A) Modelado visual
- B) Desarrollo iterativo
- C) Control de cambios
- D) Gestión documental

11. ¿Cuál es una desventaja del desarrollo incremental?

- A) Entrega rápida
- B) Mayor visibilidad del proceso
- C) Degradación de la arquitectura
- D) Retroalimentación continua

12. ¿Qué tipo de sistema se adapta mejor al modelo en cascada?

- A) Sistema de redes sociales
- B) Aplicación de mensajería móvil
- C) Sistema de control de tráfico aéreo
- D) Tienda en línea

13. ¿Qué se busca con la validación de requisitos?

- A) Crear casos de prueba
- B) Asegurar que los requisitos sean completos y realistas
- C) Asignar tareas al equipo
- D) Identificar errores de código

14. ¿Qué es el diseño arquitectónico?

- A) Creación del logotipo del software
- B) Organización de reuniones
- C) Identificación de componentes principales y su relación
- D) Manual de instalación

15. ¿Cuál es una actividad del flujo de trabajo en RUP?

- A) Redacción de contratos
- B) Modelado del negocio
- C) Auditoría contable
- D) Análisis financiero

16. ¿Cuál es una buena práctica del RUP?

- A) Gestión económica
- B) Diseño exclusivo en papel
- C) Control de cambios
- D) Validación jurídica

17. ¿Qué se hace en la fase de transición del RUP?

- A) Evaluar riesgos
- B) Elaborar prototipos
- C) Entregar el software al usuario final
- D) Diseñar interfaces

18. ¿Qué son las precondiciones en un proceso?

- A) Normas después de realizar la tarea
- B) Reglas antes de realizar una actividad
- C) Indicadores de calidad
- D) Errores comunes

19. ¿Qué ventaja tiene la reutilización de software?

- A) Aumenta el tiempo de desarrollo
- B) Reduce riesgos y costos
- C) Exige programación desde cero
- D) Elimina la fase de pruebas

20. ¿Qué función tienen las pruebas de aceptación?

- A) Revisar los errores de los programadores
- B) Verificar compatibilidad con el hardware
- C) Validar que el sistema cumple con los requisitos del cliente
- D) Identificar errores gramaticales

21. ¿Qué se entiende por evolución del software?

Respuesta: Adaptación continua del sistema a nuevos requisitos o necesidades del cliente.

22. Menciona dos tipos de componentes reutilizables.

Respuesta: Servicios web y colecciones de objetos.

23. ¿En qué consiste la validación del software?

Respuesta: En asegurar que el software cumple con los requisitos y expectativas del cliente.

24. ¿Qué es el RUP y qué modelo combina?

Respuesta: Es el Proceso Unificado Racional; combina aspectos de modelos como cascada, incremental y reutilización.

25. Menciona una ventaja del desarrollo incremental.

Respuesta: Permite entregas tempranas y retroalimentación continua del cliente.

Métodos de desarrollo ágil

Las metodologías de desarrollo ágil son enfoques iterativos e incrementales que buscan mejorar la eficiencia y la flexibilidad en la creación de software. Se centran en la colaboración continua con el cliente y en la adaptación a requisitos cambiantes, priorizando la entrega temprana y continua de software funcional.

A continuación, se detallan las principales actividades involucradas en las metodologías ágiles, junto con ejemplos ilustrativos:

1. **Comunicación con el cliente**: La interacción constante con el cliente es fundamental para comprender sus necesidades y expectativas. Por ejemplo, en Scrum, el Product Owner representa al cliente y trabaja estrechamente con el equipo para definir y priorizar las funcionalidades del producto.

2. **Planeación**: Se planifican iteraciones cortas, conocidas como sprints en Scrum, que suelen durar entre una y cuatro semanas. Durante la reunión de planificación del sprint, el equipo selecciona las tareas del backlog que se compromete a completar en esa iteración.

3. **Modelado**: Consiste en la creación de prototipos o diagramas que representen las funcionalidades del sistema. Por ejemplo, en Extreme Programming (XP), se utilizan tarjetas CRC (Clase, Responsabilidad, Colaborador) para modelar y discutir las responsabilidades de las clases en el sistema.

4. **Construcción**: Es la fase de desarrollo donde se codifican las funcionalidades planificadas. En XP, se practica la programación en parejas, donde dos desarrolladores trabajan juntos en una misma estación de trabajo para mejorar la calidad del código y compartir conocimientos.

5. **Entrega**: Al final de cada iteración, se entrega una versión funcional del software al cliente para su evaluación. Por ejemplo, en Scrum, al concluir un sprint, se realiza una reunión de revisión donde el equipo demuestra las funcionalidades completadas al cliente.

6**. Evolución**: Basándose en la retroalimentación del cliente, se realizan ajustes y mejoras en el producto. En Kanban, se visualizan las tareas en un tablero, lo que permite identificar cuellos de botella y áreas de mejora continua en el proceso de desarrollo.

A continuación, se presentan algunos de los principales modelos de desarrollo ágil:

1. **Programación Extrema (Extreme Programming - XP)**: XP se centra en mejorar la calidad del software y la capacidad de respuesta ante los cambios de requisitos del cliente. Entre sus prácticas destacan la programación en parejas, el desarrollo orientado a pruebas y la integración continua.

Esquema

Metodología XP – Programación Extrema

La imagen anterior muestra un esquema del ciclo de desarrollo en la Metodología XP (Programación Extrema), un enfoque ágil que se basa en iteraciones cortas, retroalimentación continua y colaboración estrecha entre desarrolladores y clientes.

Explicación del Ciclo de XP

El diagrama presenta las fases clave en el desarrollo de software con XP, representadas en un ciclo iterativo:

Planificación:

- Se recopilan las historias de usuario, que describen los requisitos del sistema desde la perspectiva del usuario final.
- Se establecen los valores y criterios de pruebas de adaptación.
- Se define un plan de iteración, en el cual se establece qué funcionalidad se desarrollará en el siguiente ciclo.

Diseño:

- Se trabaja en un diseño simple, evitando la complejidad innecesaria.
- Se utilizan herramientas como tarjetas CRC (Class-Responsibility-Collaborator) para modelar el sistema.
- Se crean prototipos y se buscan soluciones en puntos clave del desarrollo.

Codificación:

- Se implementa el código con un enfoque en la calidad y la colaboración.
- Se emplea programación en parejas, donde dos desarrolladores trabajan en conjunto para mejorar la calidad del código.
- Se realizan rediseños según sea necesario para mejorar la estructura del software.

Pruebas:

- Se ejecutan pruebas unitarias y se aplica integración continua para asegurar que cada parte del código funcione correctamente.
- Se realizan pruebas de adaptación para validar que el sistema se ajusta a los requerimientos del usuario.

Lanzamiento:

- Se entrega un incremento del software, es decir, una nueva versión funcional del sistema.
- Se mide la velocidad calculada del proyecto para evaluar el progreso y planificar futuras iteraciones.

Ciclo Iterativo

XP sigue un enfoque iterativo y adaptativo, lo que significa que, después de cada lanzamiento, el proceso vuelve a la planificación para definir nuevas funcionalidades y mejoras. Esto permite un desarrollo ágil y flexible, donde el software evoluciona continuamente en respuesta a las necesidades del usuario.

Este conjunto de valores es clave en la metodología XP, ya que permite que los equipos de desarrollo trabajen de manera eficiente y colaborativa, asegurando la entrega de software de alta calidad.

Los valores principales que se destacan en el diagrama son:

Simplicidad (en el centro del diagrama): Se refiere a la idea de mantener el código y el desarrollo lo más simple posible, evitando complejidad innecesaria.

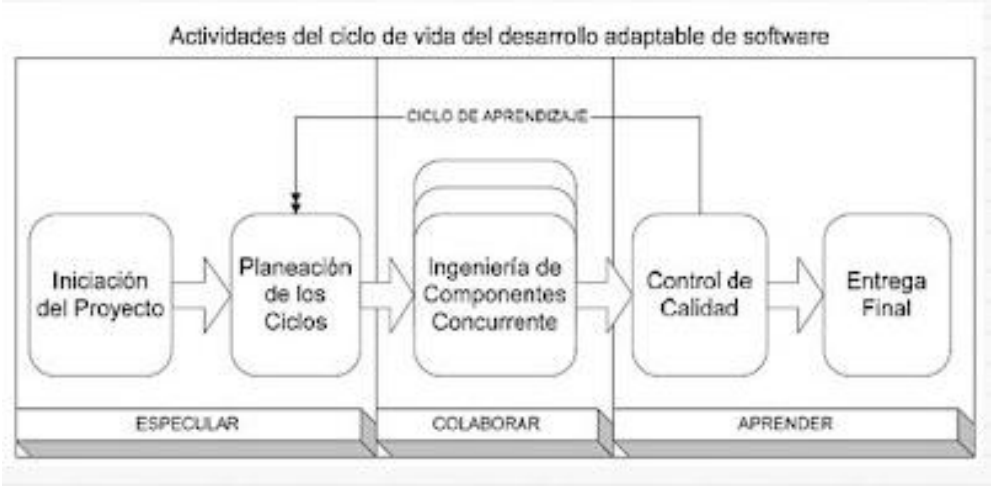
Comunicación: Fomenta la interacción constante entre los miembros del equipo para asegurar que todos comprendan los objetivos y el progreso del proyecto.

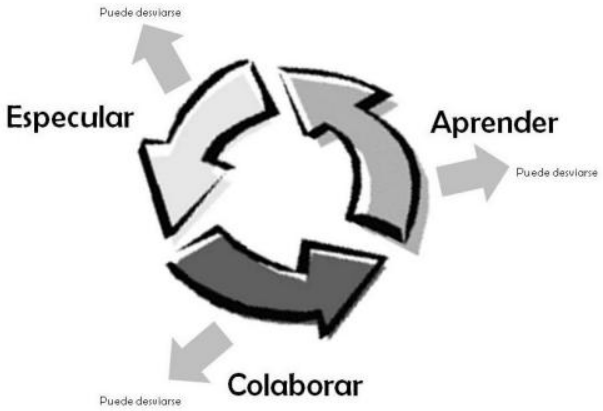
Respeto: Promueve un ambiente de trabajo donde cada miembro valora el esfuerzo y las ideas de los demás.

Retroalimentación (Feedback): Es esencial en XP para realizar mejoras continuas, ya sea a través de pruebas, revisiones de código o interacciones con los clientes.

Coraje: Se refiere a la capacidad de enfrentar desafíos, realizar cambios necesarios y mejorar continuamente el proceso de desarrollo.

2. **Desarrollo Adaptativo de Software (Adaptive Software Development - ASD):** ASD enfatiza la adaptabilidad y la colaboración en equipos autoorganizados. El ciclo de vida de ASD consta de tres fases: especulación, colaboración y aprendizaje, permitiendo una rápida adaptación a los cambios y la entrega continua de valor al cliente.





La imagen anterior muestra un diagrama sobre las actividades del ciclo de vida del desarrollo adaptable de software, lo cual está relacionado con metodologías ágiles.

Explicación del Diagrama

Este diagrama divide el ciclo de vida del desarrollo en varias fases principales, organizadas en tres enfoques clave:

Especular: Relacionado con la planificación y la visión del proyecto.

- Iniciación del Proyecto: Se define el alcance, objetivos y requisitos generales del software.
- Planeación de los Ciclos: Se establecen las iteraciones o ciclos en los que se desarrollará el software.

Colaborar: Enfocado en la construcción y desarrollo del software.

- Ingeniería de Componentes Concurrente: Desarrollo del software con una arquitectura flexible y adaptable, trabajando en paralelo con múltiples componentes.

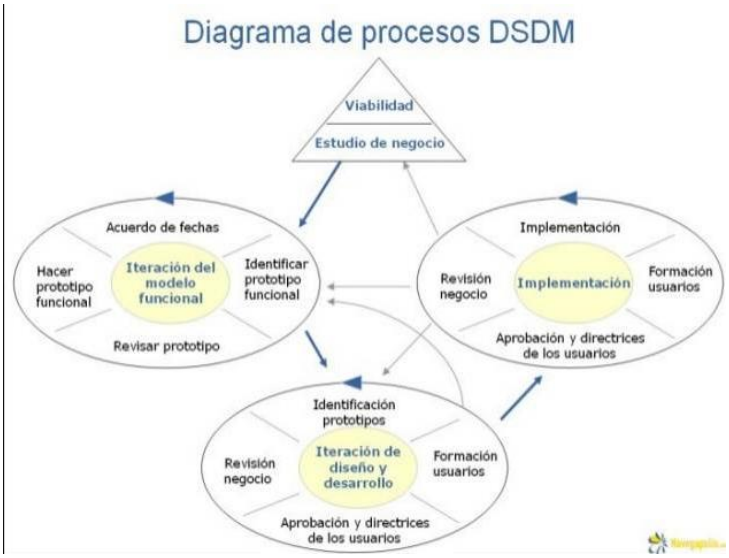
Aprender: Implica la validación y mejora continua del software.

- Control de Calidad: Se realizan pruebas y validaciones para garantizar el correcto funcionamiento del software.
- Entrega Final: Se finaliza y entrega el producto, asegurando que cumpla con los requisitos establecidos.

Ciclo de Aprendizaje

El diagrama indica que existe un ciclo de aprendizaje, lo que significa que hay retroalimentación constante entre las fases. Esto permite hacer ajustes en la planificación y el desarrollo para mejorar la calidad y adaptabilidad del software.

3. **Método de Desarrollo de Sistemas Dinámicos (Dynamic Systems Development Method - DSDM):** DSDM es una metodología ágil que se centra en la entrega rápida y continua de sistemas de software, priorizando la colaboración con el cliente y la entrega de productos de alta calidad. Se basa en principios como la participación activa del usuario y la entrega frecuente de productos.



El diagrama anterior representa el proceso de desarrollo DSDM (Dynamic Systems Development Method), una metodología ágil utilizada para el desarrollo de software que enfatiza la entrega rápida y continua de productos funcionales.

Explicación del Diagrama

El proceso de DSDM se divide en varias fases clave:

Viabilidad y Estudio de Negocio:

- Se evalúa la factibilidad del proyecto y se realiza un análisis del negocio para entender los requisitos y necesidades de los usuarios.

Iteración del Modelo Funcional:

- Se crean prototipos funcionales para visualizar y validar la funcionalidad inicial del sistema.
- Se identifican los prototipos que representan el sistema en desarrollo.
- Se revisan los prototipos con los usuarios y otras partes interesadas para asegurar que se ajusten a los requisitos.
- Se acuerdan fechas y tiempos para las siguientes iteraciones.

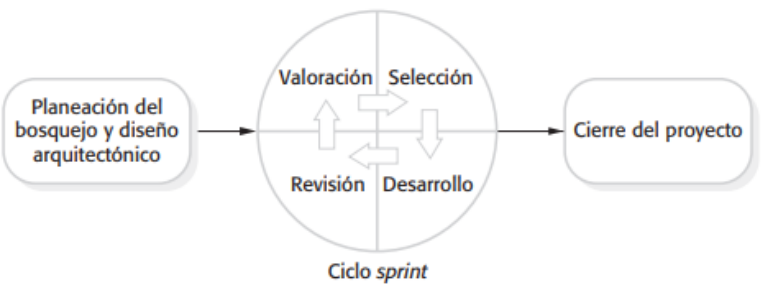
Iteración de Diseño y Desarrollo:

- Se seleccionan los prototipos a desarrollar con mayor profundidad.
- Se mejoran y refinan los prototipos mediante iteraciones, asegurando que cumplan con los requisitos del negocio y de los usuarios.
- Se revisan los avances con el negocio y los usuarios para obtener aprobación y directrices adicionales.
- Se proporciona formación a los usuarios finales para que puedan comprender cómo interactuar con el sistema.

Implementación:

- Se realiza la entrega final del sistema desarrollado.
- Se lleva a cabo una revisión del negocio para validar que el software cumple con los objetivos.
- Se obtiene la aprobación final de los usuarios antes de poner el sistema en producción.
- Se ofrece formación a los usuarios para garantizar una adopción efectiva del software.

4. **Scrum:** Scrum es un marco de trabajo que facilita la colaboración en equipos multifuncionales y autoorganizados. Se basa en ciclos de trabajo llamados "sprints", que suelen durar entre una y cuatro semanas, al final de los cuales se entrega un incremento funcional del producto.



El diagrama anterior representa un ciclo de desarrollo ágil basado en Sprints, un concepto fundamental en Scrum, una metodología ágil para la gestión de proyectos.

Explicación del Diagrama

Planeación del bosquejo y diseño arquitectónico:

- En esta fase inicial, se definen los requisitos generales del proyecto y se establece una visión del producto.
- Se diseñan las bases arquitectónicas y estructurales necesarias para el desarrollo del software o sistema.

Ciclo Sprint (Iteraciones en el desarrollo):

Este ciclo es repetitivo e iterativo, asegurando mejoras constantes en cada fase. Se compone de cuatro actividades clave:

- Valoración: Se evalúan las funcionalidades, su prioridad y viabilidad.
- Selección: Se eligen las características o tareas específicas que se desarrollarán en la iteración.
- Desarrollo: Se implementan y codifican las funcionalidades seleccionadas.
- Revisión: Se prueba y se valida el resultado con retroalimentación para mejorar el producto en la siguiente iteración.

Cierre del Proyecto:

- Cuando todas las iteraciones han completado los objetivos definidos, se realiza la entrega final del producto.
- Se revisa el cumplimiento de los requisitos y se finaliza oficialmente el desarrollo.

5. **Cristal (Crystal):** Crystal es una familia de metodologías ágiles que se adapta al tamaño del equipo y la criticidad del proyecto. Cada variante de Crystal (como Crystal Clear, Crystal Orange) se ajusta a diferentes contextos, enfatizando la comunicación, la simplicidad y la reflexión periódica.

La imagen anterior representa los niveles de la metodología ágil Crystal, una familia de metodologías de desarrollo de software que se adapta según el tamaño del equipo y la complejidad del proyecto.

Explicación del Diagrama

La metodología Crystal clasifica los proyectos en diferentes categorías, cada una con un color y reglas específicas según el número de personas en el equipo:

Crystal Clear (Blanco/Transparente) - 0 a 8 personas

- Adecuado para equipos pequeños y proyectos con baja complejidad.
- Requiere comunicación frecuente y colaboración directa.
- Menos documentación formal y más enfoque en la interacción.

Crystal Yellow (Amarillo) - 9 a 20 personas

- Requiere una estructura más organizada.
- Comunicación fluida, pero con algunas reglas y procesos adicionales.
- Más documentación para coordinar a los miembros del equipo.

Crystal Orange (Naranja) - 21 a 50 personas

- Necesita planificación y gestión más formalizada.
- Uso de herramientas de colaboración y documentación detallada.
- Se introducen roles especializados en el equipo.

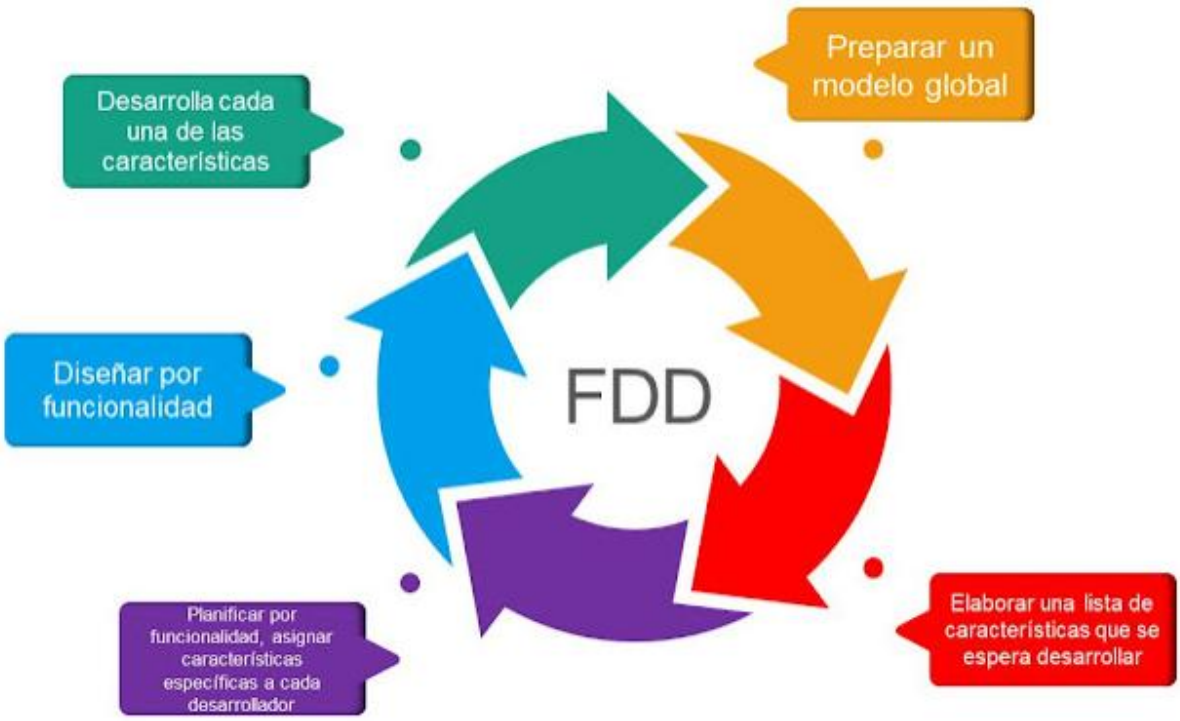
Crystal Red (Rojo) - 51 a 100 personas

- Aplicado en proyectos más grandes y complejos.
- Uso de procesos más estructurados y herramientas avanzadas de gestión.
- Necesidad de dividir el trabajo en subequipos.

Crystal Diamond o Sapphire (Azul) - Proyectos Especiales

- Aplicado en proyectos críticos (por ejemplo, sistemas bancarios, aeronáuticos o médicos).
- Requiere altos estándares de calidad y seguridad.
- Aplicación de metodologías más estrictas y control exhaustivo.

6. **Desarrollo Guiado por Funcionalidades (Feature-Driven Development - FDD):** FDD se centra en el diseño y construcción de funcionalidades específicas del sistema. Sigue un proceso de cinco pasos que incluye el desarrollo de un modelo general, la construcción de una lista de funcionalidades, la planificación por funcionalidad, el diseño por funcionalidad y la construcción por funcionalidad.



La imagen anterior representa el ciclo de desarrollo en la metodología ágil FDD (Feature-Driven Development), que se centra en el desarrollo basado en características. FDD es una metodología ágil utilizada para gestionar y desarrollar software de manera iterativa y enfocada en la funcionalidad.

Explicación del Ciclo de FDD

- El proceso de FDD consta de cinco fases principales, representadas en la imagen:

Preparar un modelo global (amarillo)

- Se define una visión general del sistema mediante un modelo conceptual.
- Se identifican los principales requerimientos y reglas del negocio.

Elaborar una lista de características (rojo)

- Se crean listas de pequeñas funcionalidades o características que el sistema debe desarrollar.
- Estas características son breves y se desarrollan en ciclos cortos (entre 2 y 10 días).

Planificar por funcionalidad (morado)

- Se priorizan las características y se asignan a los desarrolladores según sus habilidades.
- Se organizan los equipos y se establecen los plazos de entrega.

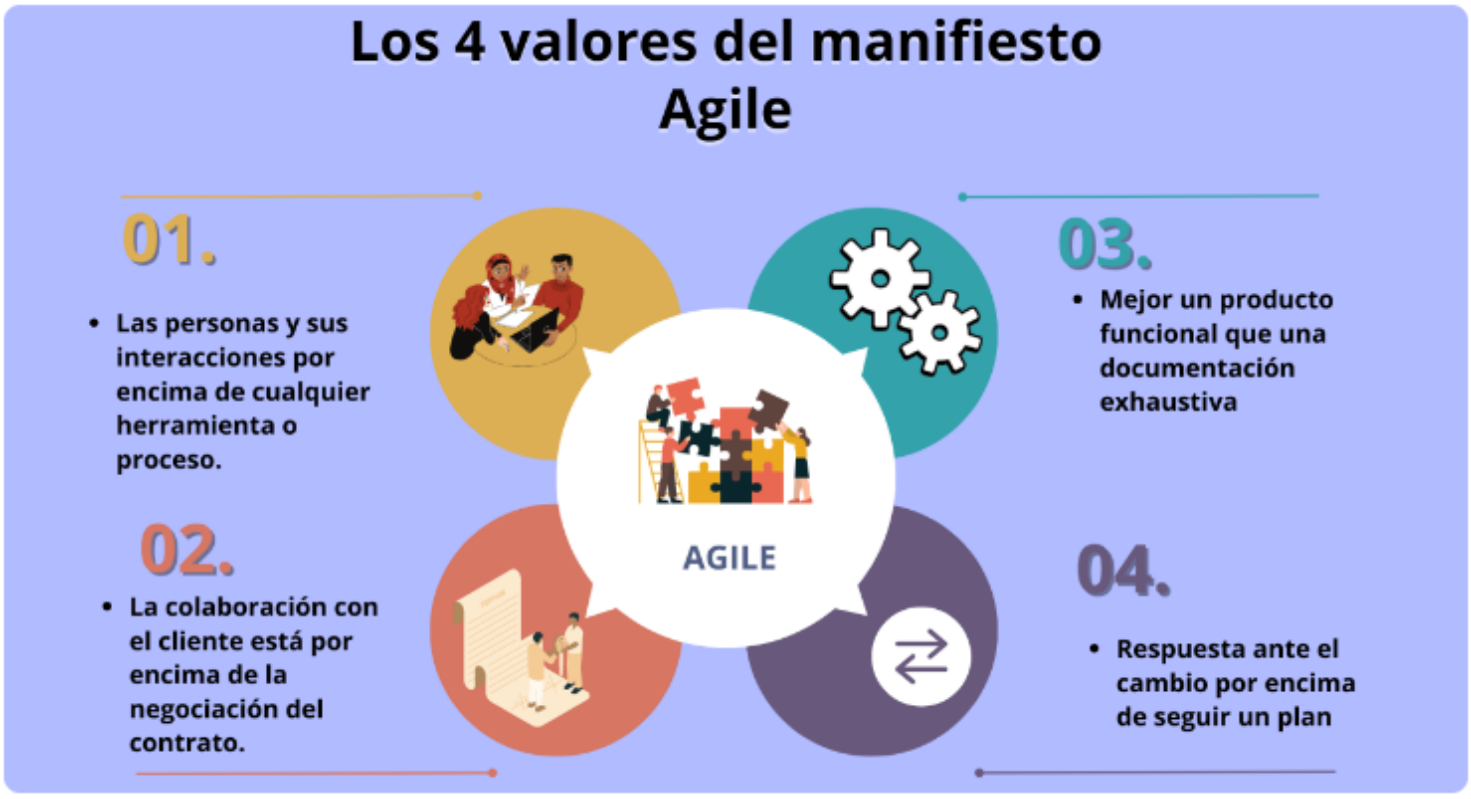
Diseñar por funcionalidad (azul)

- Se desarrolla el diseño detallado de cada característica antes de programarla.
- Se realizan revisiones para garantizar que el diseño sea óptimo antes de la implementación.

Desarrollar cada una de las características (verde)

- Se codifican, prueban y verifican las características.
- Se integran al sistema y se evalúan continuamente para asegurar calidad y funcionalidad.

7. **Modelo Ágil:** El modelo ágil es un enfoque general que engloba diversas metodologías ágiles. Se basa en el Manifiesto Ágil, que destaca valores como la colaboración con el cliente, la respuesta ante el cambio y la entrega continua de software funcional.



La imagen anterior representa los 4 valores del Manifiesto Ágil, un documento fundamental para el desarrollo de software ágil, creado en 2001 por un grupo de expertos en desarrollo de software. Estos valores guían las metodologías ágiles como Scrum, XP, Kanban y FDD, promoviendo la flexibilidad, la colaboración y la entrega rápida de software funcional.

Explicación de los 4 valores del Manifiesto Ágil

Las personas y sus interacciones están por encima de cualquier herramienta o proceso

- En lugar de enfocarse en procesos rígidos o herramientas específicas, el enfoque ágil prioriza la comunicación y la colaboración entre los miembros del equipo.
- Se fomenta el trabajo en equipo, la confianza y la adaptación en función de las necesidades del proyecto.

La colaboración con el cliente está por encima de la negociación del contrato

- En lugar de centrarse en documentos contractuales estrictos, se busca trabajar en conjunto con el cliente para adaptar el producto a sus necesidades reales.
- Se mantiene una comunicación constante con el cliente para asegurar que el software entregue valor de manera efectiva.

Mejor un producto funcional que una documentación exhaustiva

- Se prioriza entregar software que funcione en lugar de dedicar tiempo excesivo a documentación detallada que puede volverse obsoleta rápidamente.
- Esto no significa eliminar la documentación, sino hacerla más ligera y enfocada en lo esencial.

Respuesta ante el cambio por encima de seguir un plan

- En lugar de seguir planes rígidos y predefinidos, los equipos ágiles están preparados para adaptarse a los cambios y nuevas prioridades del proyecto.
- Se adopta una mentalidad flexible para responder rápidamente a cambios en los requisitos del cliente o el entorno del negocio.

Planes y desarrollos ágiles

Los planes y desarrollos ágiles se basan en la flexibilidad, la iteración continua y la colaboración estrecha con el cliente. A diferencia de los enfoques tradicionales como el modelo en cascada, donde se establece un plan fijo desde el inicio, en las metodologías ágiles se permite ajustar los requerimientos y prioridades a medida que avanza el proyecto. Esto se logra a través de ciclos cortos de desarrollo llamados Sprints en Scrum o entregas incrementales en Kanban. El objetivo principal es garantizar que el software entregue valor real en cada iteración.

Por ejemplo, en un proyecto de desarrollo de una aplicación móvil de comercio electrónico, un equipo ágil podría iniciar con un Sprint de dos semanas para diseñar y programar la funcionalidad básica de inicio de sesión y catálogo de productos. En lugar de esperar a que todo el sistema esté completo, esta primera versión se lanza a un grupo de usuarios para obtener retroalimentación. Si los usuarios encuentran problemas con la experiencia de compra, el equipo puede ajustar la interfaz y mejorar la funcionalidad en el siguiente Sprint sin afectar el desarrollo de otras características.

Un aspecto clave en los planes ágiles es la capacidad de adaptación. Por ejemplo, una empresa de desarrollo de software financiero puede comenzar con una planificación inicial basada en las necesidades del cliente, pero si durante las primeras iteraciones descubren que los usuarios requieren una mayor seguridad en las transacciones, el equipo puede modificar el plan para incluir protocolos adicionales de autenticación. Esto permite que el producto final sea más útil y seguro sin retrasar innecesariamente la entrega de otras funcionalidades.

Las metodologías ágiles también fomentan la colaboración constante. Un equipo de una startup de inteligencia artificial que desarrolla un asistente virtual podría reunirse diariamente en reuniones cortas llamadas Daily Stand-up para discutir avances, bloqueos y prioridades. Esto facilita la comunicación y evita que problemas pequeños se conviertan en obstáculos mayores. Gracias a esta dinámica, el equipo puede lanzar versiones mejoradas del asistente cada pocas semanas, en lugar de esperar meses para un lanzamiento completo.

En conclusión, los planes ágiles permiten una mayor eficiencia y adaptación a las necesidades del mercado. En proyectos donde la incertidumbre es alta, como el desarrollo de videojuegos o plataformas de servicios en la nube, la metodología ágil permite ajustar prioridades y mejorar continuamente la calidad del producto. Empresas como Spotify, Amazon y Google utilizan estos enfoques para innovar rápidamente, asegurando que sus productos evolucionen con las necesidades de los usuarios.

Administración de un Proyecto Ágil

La administración de proyectos ágiles de software implica la supervisión del desarrollo de software asegurando que se entregue a tiempo y dentro del presupuesto. A diferencia de los métodos tradicionales, donde la planificación detallada es clave, los métodos ágiles se basan en la flexibilidad y la adaptación continua a los cambios en los requisitos y necesidades del cliente.

Uno de los enfoques ágiles más utilizados es Scrum, que se centra en la administración iterativa de proyectos a través de ciclos llamados sprints, los cuales suelen durar entre 2 y 4 semanas. Durante estos ciclos, se desarrolla y entrega un incremento del sistema. Scrum se basa en la colaboración y la comunicación constante dentro del equipo y con el cliente.

Los sprints siguen un proceso estructurado con cinco fases clave:

Duración fija: De dos a cuatro semanas para garantizar entregas constantes.

Valoración del sprint: Se revisan las tareas y se asignan prioridades.

Selección del sprint: Se eligen las características a desarrollar en cada iteración.

Desarrollo y revisión: Se trabaja en las funcionalidades y se realizan reuniones diarias para evaluar avances y resolver problemas.

Cierre del sprint: Se presenta el trabajo completado a los participantes para recibir retroalimentación.

El Scrum Master facilita el proceso organizando reuniones diarias, rastreando el progreso y asegurando que no haya bloqueos en el desarrollo. A diferencia de los métodos tradicionales, Scrum empodera a los equipos para tomar decisiones sin depender de un administrador centralizado.

Entre los beneficios de Scrum, se destacan:

- Descomposición del producto en unidades pequeñas y manejables.
- Adaptabilidad a requisitos cambiantes sin afectar el progreso.
- Mayor comunicación y sincronización entre el equipo.
- Retroalimentación continua por parte del cliente.
- Creación de una cultura colaborativa donde todos trabajan hacia el éxito del proyecto.

Scrum nació con la idea de aplicarse en equipos co-localizados, pero con la evolución del software y la globalización, también se ha adaptado a equipos distribuidos. Su flexibilidad y eficacia lo han convertido en una metodología ampliamente utilizada en la industria del desarrollo de software.

Escalamiento de Métodos Ágiles

Los métodos ágiles fueron diseñados inicialmente para equipos pequeños que podían comunicarse de manera informal y trabajar en la misma ubicación. Sin embargo, a medida que la necesidad de entrega rápida de software creció, estos métodos comenzaron a aplicarse en sistemas más grandes, generando la necesidad de escalar la agilidad para proyectos desarrollados por grandes organizaciones.

Denning y sus colaboradores (2008) sugieren que para evitar los problemas comunes de la ingeniería de software, como sistemas que no cubren las necesidades del cliente o exceden el presupuesto, es necesario encontrar maneras de adaptar los métodos ágiles a grandes sistemas. Leffingwell (2007) documenta prácticas que han sido utilizadas con éxito en estos entornos, mientras que Moore y Spens (2008) reportan su experiencia en el desarrollo de un sistema médico con 300 desarrolladores en equipos distribuidos.

El desarrollo de grandes sistemas difiere del de sistemas pequeños en varios aspectos:

1. **Sistemas separados**: Los grandes sistemas suelen estar conformados por múltiples módulos independientes desarrollados por equipos separados, lo que dificulta una visión unificada.
2. **Sistemas heredados**: Muchos proyectos deben integrarse con sistemas existentes, lo que impone restricciones al desarrollo ágil.
3. **Integración compleja**: A menudo, el trabajo principal no es desarrollar software desde cero, sino configurar y actualizar sistemas existentes.

4. **Restricciones normativas**: Grandes organizaciones tienen regulaciones y estándares estrictos que limitan la flexibilidad del desarrollo ágil.
5. **Largos tiempos de desarrollo**: Mantener equipos cohesionados durante períodos largos es difícil debido a la rotación de personal.
6. **Diversidad de participantes**: En proyectos grandes, hay múltiples partes interesadas con diferentes necesidades y prioridades.

Para abordar estos desafíos, existen dos enfoques principales para escalar la agilidad:

- **Scaling Up (Expansión**): Aplicar métodos ágiles en el desarrollo de grandes sistemas de software que no pueden desarrollarse con equipos pequeños.
- **Scaling Out (Ampliación)**: Integrar la agilidad en organizaciones grandes con estructuras tradicionales.

Leffingwell (2007) propone algunas adaptaciones clave para escalar la agilidad:

- Incorporar documentación estructurada y diseño arquitectónico sin perder la flexibilidad.
- Establecer mecanismos de comunicación eficaces, como videoconferencias y plataformas colaborativas.
- Implementar integración continua, asegurando que los diferentes módulos del sistema puedan fusionarse fácilmente.

Las grandes empresas enfrentan desafíos adicionales al adoptar métodos ágiles, como la falta de experiencia en metodologías ágiles por parte de los gerentes, la resistencia organizacional debido a estándares burocráticos y dificultades en la gestión del cambio. Introducir y sostener la agilidad a nivel organizacional requiere un cambio cultural, promovido por líderes internos que impulsen la transformación.

En conclusión, aunque los métodos ágiles fueron creados para equipos pequeños, su escalamiento en grandes organizaciones es posible con adaptaciones estratégicas. Empresas que han logrado esta transición han demostrado que la agilidad puede mejorar la entrega de software en entornos complejos y altamente regulados.

Métodos De Desarrollo De Software

Los métodos de desarrollo de software han evolucionado para adaptarse a las necesidades cambiantes de los proyectos y las organizaciones. Dos enfoques destacados en este ámbito son el desarrollo iterativo y el desarrollo incremental.

Desarrollo Iterativo:

El desarrollo iterativo es un proceso en el cual el software se construye y mejora a través de repetidas iteraciones o ciclos. Cada ciclo implica la planificación, diseño, implementación y evaluación de una versión del software, permitiendo refinamientos continuos basados en la retroalimentación obtenida. Este enfoque facilita la adaptación a cambios en los requisitos y mejora la calidad del producto final.

Características principales:

Ciclos repetitivos: El proyecto se divide en iteraciones que abarcan todas las fases del desarrollo, desde el análisis hasta la implementación y pruebas.

Retroalimentación continua: Cada iteración proporciona una oportunidad para evaluar el progreso y realizar ajustes según sea necesario.

Flexibilidad: Permite incorporar cambios y mejoras de manera continua durante el proceso de desarrollo.

Ventajas:

Detección temprana de problemas: Las iteraciones frecuentes permiten identificar y resolver inconvenientes en etapas tempranas del desarrollo.

Mejora continua: La retroalimentación constante contribuye a la evolución y optimización del producto.

Desventajas:

Complejidad en la gestión: Requiere una planificación y seguimiento detallados para coordinar las iteraciones y mantener la coherencia del proyecto.

Desarrollo Incremental:

El desarrollo incremental consiste en dividir el proyecto en partes más pequeñas y manejables, conocidas como incrementos. Cada incremento añade funcionalidad al sistema de forma progresiva hasta completar el producto final. Este enfoque permite entregar versiones funcionales del software en etapas tempranas, lo que facilita su evaluación y uso por parte de los usuarios.

Características principales:

División en incrementos: El proyecto se segmenta en módulos o componentes que se desarrollan y entregan de manera secuencial.

Entrega progresiva: Cada incremento proporciona una versión funcional del software con nuevas características o mejoras.

Priorización de funcionalidades: Las funcionalidades más importantes o de mayor valor para el usuario se desarrollan en los primeros incrementos.

Ventajas:

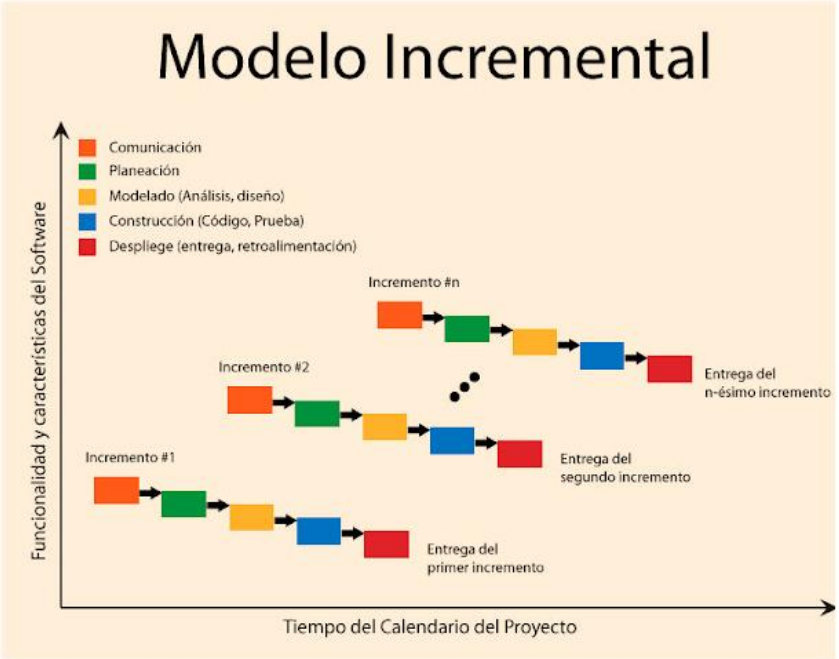
Entrega temprana de valor: Los usuarios pueden comenzar a utilizar partes funcionales del software antes de que se complete el desarrollo total.

Reducción de riesgos: La implementación gradual permite identificar y mitigar riesgos de manera oportuna.

Desventajas:

Integración compleja: La incorporación de nuevos incrementos puede generar desafíos en la integración con las partes existentes del sistema.

Necesidad de una arquitectura sólida: Es fundamental contar con una arquitectura de software que soporte la adición progresiva de funcionalidades sin comprometer la estabilidad del sistema.



Requerimientos

Los requerimientos en el desarrollo de software son las condiciones o capacidades que un sistema debe cumplir para satisfacer las necesidades de los usuarios o clientes. Estos sirven como base para el diseño, desarrollo y validación del software, asegurando que el producto final cumpla con las expectativas establecidas.

Tipos de Requerimientos:

Requerimientos Funcionales: Describen las funciones específicas que el sistema debe realizar. Incluyen las tareas, datos y comportamientos que el software debe ejecutar en respuesta a entradas específicas. Por ejemplo, en un sistema bancario, un requerimiento funcional podría ser "permitir transferencias de fondos entre cuentas de clientes".

Requerimientos No Funcionales: Se refieren a las características de calidad que el sistema debe poseer, como rendimiento, seguridad, usabilidad y confiabilidad. Por ejemplo, "el sistema debe procesar 1000 transacciones por segundo" es un requerimiento no funcional relacionado con el rendimiento.

Especificaciones:

La Especificación de Requisitos de Software (ERS) o (SRS) es un documento que detalla de manera completa y clara todos los requerimientos del sistema. Incluye descripciones de las funcionalidades, restricciones y criterios de calidad que el software debe cumplir. Una buena ERS debe ser completa, consistente, inequívoca, verificable y modificable.

Adquisición de Requerimientos:

También conocida como elicitación, esta fase implica recopilar información de las partes interesadas para comprender sus necesidades y expectativas. Se utilizan técnicas como entrevistas, encuestas, talleres y observación para obtener una comprensión detallada de lo que se espera del sistema.

Análisis de Requerimientos:

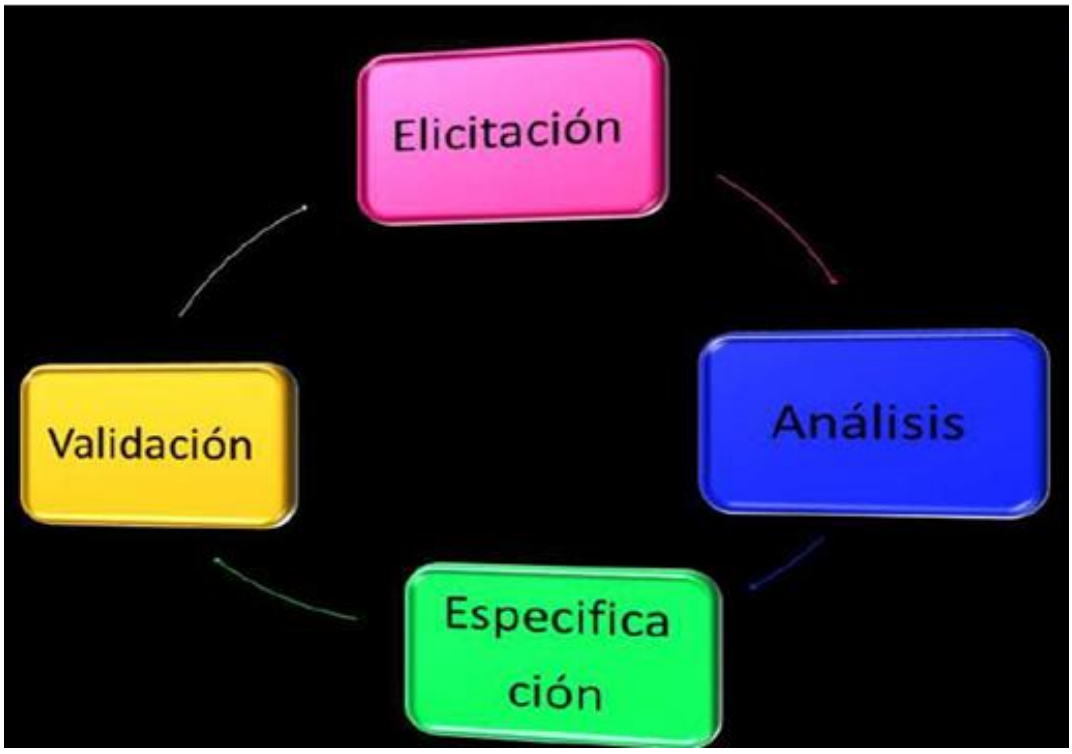
En esta etapa, se evalúan y refinan los requerimientos recopilados para garantizar que sean claros, completos y factibles. El análisis ayuda a identificar posibles conflictos o inconsistencias y asegura que los requerimientos estén alineados con los objetivos del negocio.

Validación de Requerimientos:

La validación consiste en verificar que los requerimientos definidos reflejen correctamente las necesidades y expectativas de los usuarios. Se realizan revisiones, prototipos y pruebas para asegurarse de que el sistema desarrollado cumplirá con los requerimientos especificados.

Administración de Requerimientos:

Esta disciplina se encarga de documentar, analizar, rastrear y priorizar los requerimientos a lo largo del ciclo de vida del proyecto. La gestión efectiva de los requerimientos es crucial para adaptarse a cambios y garantizar que el producto final cumpla con las necesidades del cliente.



Cuestionario: Métodos Ágiles y Documentación de Software

1. ¿Cuál es el objetivo principal de las metodologías ágiles?

A) Reducir los costos de hardware

★ B) Mejorar la eficiencia y adaptabilidad en el desarrollo de software

C) Documentar detalladamente cada fase del proyecto

D) Eliminar completamente la planificación inicial

2. En Scrum, ¿quién representa al cliente y prioriza las funcionalidades del producto?

A) Scrum Master

★ B) Product Owner

C) Team Leader

D) Analista de calidad

3. ¿Qué práctica es característica de la Programación Extrema (XP)?

★ A) Programación en parejas

B) Desarrollo en cascada

C) Revisión mensual de requisitos

D) Uso de esquemas de Gantt

4. ¿Qué valor NO pertenece al Manifiesto Ágil?

A) Personas e interacciones sobre procesos y herramientas

B) Respuesta ante el cambio sobre seguir un plan

★ C) Documentación exhaustiva sobre producto funcional

D) Colaboración con el cliente sobre negociación contractual

5. En la metodología ASD, la fase de "Aprender" se asocia principalmente con:

A) Definición de objetivos

★ B) Validación y mejora continua

C) Construcción del modelo global

D) Selección de funcionalidades

6. ¿Qué enfoque utiliza Kanban para visualizar el flujo de trabajo?

A) Prototipado rápido

★ B) Uso de tableros

C) Diseño modular

D) Diagramas de casos de uso

7. En DSDM, ¿cuál es la función principal de los prototipos funcionales?

A) Servir como documentación legal

B) Asegurar la integración con sistemas heredados

★ C) Visualizar y validar la funcionalidad del sistema

D) Garantizar seguridad informática

8. ¿Qué metodología se adapta según el tamaño del equipo y la complejidad del proyecto?

A) Scrum

B) XP

★ C) Crystal

D) FDD

9. En el desarrollo iterativo, ¿cuál es una de sus principales ventajas?

★ A) Detección temprana de problemas

B) Eliminación del análisis de requisitos

C) Uso exclusivo de documentación física

D) Finalización rápida sin retroalimentación

10. ¿Cuál de las siguientes es una característica del desarrollo incremental?

A) Se completa todo el sistema antes de realizar entregas

B) No se permite cambiar prioridades durante el proyecto

★ C) Se entrega funcionalidad de forma progresiva

D) Requiere menos planificación que el desarrollo en cascada

11. ¿Qué fase del ciclo XP se enfoca en mantener el diseño simple y evitar complejidad innecesaria?

★ A) Diseño

B) Planeación

C) Codificación

D) Lanzamiento

12. ¿Cuál es una característica de los equipos en la metodología Crystal Clear?

A) Requieren documentación extensa

B) Se organizan por departamentos

★ C) Son equipos pequeños con comunicación directa

D) Están compuestos por más de 50 personas

13. ¿Qué rol cumple el Scrum Master en un proyecto ágil?

A) Representa al cliente

★ B) Facilita el proceso y elimina bloqueos

C) Diseña la interfaz de usuario

D) Evalúa el producto final

14. ¿Qué práctica es clave en XP para asegurar que el sistema cumple con los requisitos del usuario?

A) Diagramas UML

★ B) Pruebas de adaptación

C) Análisis de regresión

D) Diseño estructurado

15. ¿Qué busca el modelo ágil frente a métodos tradicionales como el modelo en cascada?

A) Generar documentación exhaustiva

★ B) Flexibilidad y adaptación continua

C) Mantener un plan rígido

D) Finalizar el producto sin cambios

16. ¿Cuál es una de las fases del ciclo de vida en el Desarrollo Adaptativo de Software (ASD)?

A) Validación de código

B) Construcción en línea

★ C) Especulación

D) Documentación exhaustiva

17. ¿Qué caracteriza a los incrementos en el desarrollo incremental?

A) Cada incremento es probado por separado pero no integrado

B) No aportan funcionalidad al usuario

★ C) Añaden funcionalidades de manera progresiva

D) Son fases no funcionales del sistema

18. En FDD, ¿qué paso sigue después de elaborar la lista de características?

A) Implementación general

★ B) Planificar por funcionalidad

C) Codificación global

D) Verificación del sistema

19. ¿Qué se obtiene al final de cada Sprint en Scrum?

A) Un manual de usuario

★ B) Un incremento funcional del producto

C) Un prototipo sin valor funcional

D) Un informe de retroalimentación solamente

20. ¿Qué enfoque propone Leffingwell para escalar métodos ágiles en grandes organizaciones?

A) Evitar la documentación formal

★ B) Incorporar diseño estructurado sin perder flexibilidad

C) Mantener todos los equipos co-localizados

D) Prohibir el uso de integración continua

Técnicas preventivas para fortalecer el autocontrol y el bienestar mental

Autocontrol y salud mental

El autocontrol es la capacidad de regular y gestionar nuestras emociones, impulsos y comportamientos para alcanzar metas a largo plazo y mantener el equilibrio emocional. En medicina se considera una habilidad clave para la salud mental, ya que facilita enfrentar situaciones estresantes y evita conductas perjudiciales. De igual forma, la autorregulación emocional (manejar y responder a las emociones eficazmente) es crucial para el bienestar psicológico. De hecho, un buen autocontrol emocional se asocia con relaciones interpersonales más saludables, mejor rendimiento escolar o laboral y mayor resiliencia ante el estrés.

¿Cómo se pierde y recupera el autocontrol?

La imagen muestra a una persona manifestando estrés intenso y pérdida de la calma. En la vida diaria, el autocontrol tiende a debilitarse bajo presión excesiva o situaciones adversas. Por ejemplo, eventos como un examen importante o un conflicto laboral pueden hacernos perder los nervios momentáneamente. Investigaciones señalan que niveles elevados de estrés o ansiedad actúan como detonantes que socavan la función ejecutiva del cerebro, conduciendo a reacciones más impulsivas. Además, este proceso suele volverse cíclico: la tensión interna creciente genera impulsos por alivio momentáneo (p.ej. comer compulsivamente o estallar emocionalmente) y luego sobreviene la culpa o la frustración, reavivando el estrés inicial. Sin embargo, el autocontrol puede recuperarse con técnicas deliberadas. Pausar conscientemente y respirar profundamente, por ejemplo, ayuda a regular la sobreactivación fisiológica causada por la ansiedad. De igual modo, enfocarse en realizar pequeñas acciones concretas del día a día (como ordenar el escritorio o contestar un correo pendiente) genera sensación de logro y restablece la sensación de control psicológico. Estas pausas reflexivas y hábitos de recuperación cortan el círculo vicioso del estrés y permiten volver a responder de forma serena.

Estrategias y técnicas prácticas para mantener el autocontrol

Para fortalecer el autocontrol se emplean diversas técnicas avaladas por la psicología. Entre las más efectivas se incluyen:

Respiración y relajación muscular: Aprender a realizar respiraciones profundas y pausadas (por ejemplo, inhalar lenta y profundamente con el abdomen) induce la llamada respuesta de relajación, que reduce la presión arterial y la frecuencia cardíaca. Ejercicios de relajación progresiva (tensionar y luego relajar grupos

musculares) disminuyen la tensión física acumulada. En conjunto, estas prácticas contrarrestan el estrés corporal y aportan una mayor sensación de calma y control.

Reestructuración cognitivo-conductual: Consiste en identificar y modificar pensamientos automáticos distorsionados o catastróficos. La terapia cognitiva-conductual entrena a hacerse consciente de estos patrones mentales contraproducentes, sustituyéndolos por evaluaciones más realistas y adaptativas. Esto permite afrontar los problemas emocionales con más claridad y responder a situaciones difíciles de forma más eficaz. Por ejemplo, en vez de asumir que "todo saldrá mal" en un proyecto, se aprende a plantearse preguntas sobre la evidencia real de esa creencia, evitando así la ansiedad anticipatoria.

Planificación personal y control del entorno: Organizar el tiempo y las metas personales ayuda a reducir el estrés y la impulsividad. Utilizar herramientas como listas de tareas, calendarios y objetivos SMART (específicos y temporales) optimiza la productividad y mejora el equilibrio entre vida personal y profesional . Además, modificar el ambiente físico previene conductas impulsivas. Por ejemplo, eliminar estímulos desencadenantes (como sacar la televisión de la sala de estudio) o aplicar restricciones físicas sencillas (p. ej. usar guantes para no comerse las uñas) bloquea las tentaciones antes de que surjan. Establecer rutinas regulares (horarios fijos para dormir, comer y trabajar) refuerza la sensación de control sobre el día a día.

Hábitos saludables (ejercicio y descanso): Mantener una rutina de vida saludable fortalece la capacidad de autocontrol. El ejercicio aeróbico aumenta la liberación de endorfinas (neurotransmisores que generan sensación de bienestar) y alivia el estrés diario. De hecho, la actividad física habitual no solo libera tensión, sino que simula de forma controlada la respuesta al estrés, entrenando al organismo para afrontar mejor futuras demandas. A su vez, un sueño reparador y una alimentación equilibrada estabilizan las emociones y aumentan la claridad mental. En conjunto, estas prácticas contrarrestan los efectos negativos del estrés crónico y facilitan la toma de decisiones conscientes.

La práctica de mindfulness o meditación se enfoca en prestar atención plena al momento presente. Al sentarse en calma, como muestra la imagen, se entrena la mente para observar sin juzgar los pensamientos y sensaciones. Estudios indican que el mindfulness eleva la atención y el autocontrol, al mismo tiempo que reduce los niveles de estrés y ansiedad. Por ejemplo, se recomienda realizar ejercicios simples en cualquier edad: observar conscientemente la respiración (contar cómo sube y baja el pecho) o explorar sensaciones corporales

(escuchar, sentir texturas y olores) ayudan a permanecer en el aquí y ahora. Con la práctica regular de estas técnicas de atención plena, se aprende a manejar mejor las emociones antes de que escalen descontroladamente.

Seguridad alimentaria, nutrición y desarrollo sostenible: acciones globales

Definición de pobreza y hambre

La pobreza es un fenómeno multidimensional que implica privaciones económicas y sociales básicas: falta de ingresos suficientes para cubrir necesidades de alimentación, vivienda, salud y educación.

Según la ONU y el Banco Mundial, la pobreza extrema se define por ingresos diarios muy bajos (alrededor de USD 2,15 por día). El hambre se entiende como la sensación física dolorosa que causa no consumir calorías suficientes de forma regular. La FAO define el hambre crónica (subalimentación) como la persistente falta de ingesta energética adecuada.

También se habla de inseguridad alimentaria cuando las personas carecen de acceso constante a alimentos inocuos y nutritivos suficientes.

Estrategias globales para erradicar la pobreza

Los Objetivos de Desarrollo Sostenible (ODS) de la ONU, en particular el ODS1 “Fin de la pobreza”, marcan el compromiso mundial para erradicar la pobreza extrema. Entre las estrategias se incluyen sistemas de protección social, acceso equitativo a recursos, generación de empleo, educación gratuita, salud universal, igualdad de género e inclusión económica. Organismos como el Banco Mundial apoyan estas políticas combinando crecimiento económico con protección social y acceso a servicios básicos.

Seguridad alimentaria: concepto y garantías regionales

La seguridad alimentaria existe cuando todas las personas tienen acceso físico y económico a suficientes alimentos inocuos y nutritivos para llevar una vida sana. Incluye disponibilidad, acceso, utilización y estabilidad. En América Latina y África Occidental, se implementan políticas regionales, subsidios alimentarios, proyectos de riego y fortalecimiento de cadenas locales para garantizar esta seguridad. El apoyo internacional, como el de la FAO, PMA y otras organizaciones, es fundamental.

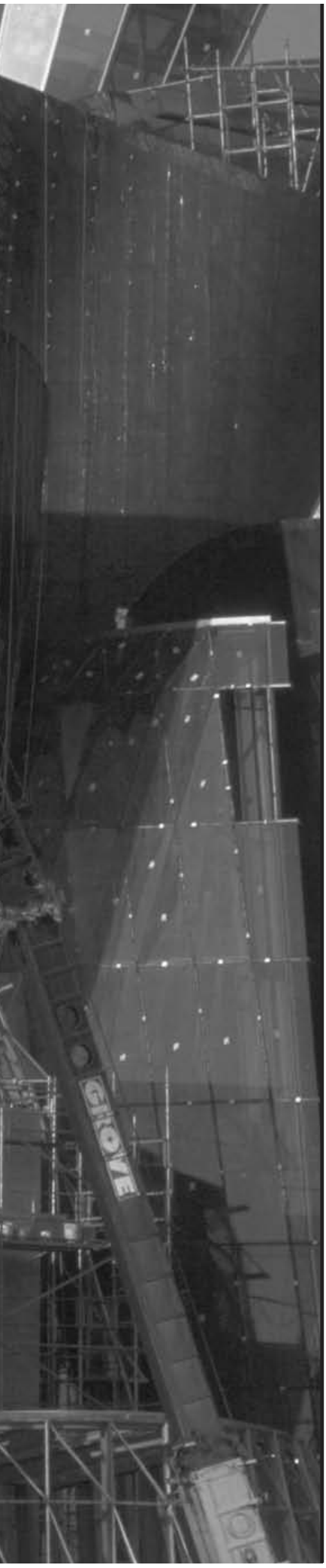
Formas de nutrición y su relevancia en salud pública global

La malnutrición incluye desnutrición y sobrepeso/obesidad. La desnutrición se manifiesta en deficiencias nutricionales y enfermedades relacionadas, especialmente en niños y embarazadas. El sobrepeso y la obesidad afectan a más del 40% de adultos a nivel mundial, elevando el riesgo de enfermedades crónicas.

Estas condiciones perpetúan el ciclo de pobreza y enfermedad, y requieren intervenciones globales en salud pública.

Agricultura sostenible: prácticas, principios y políticas

La agricultura sostenible busca satisfacer las necesidades alimentarias presentes sin comprometer los recursos del futuro. Implica prácticas como rotación de cultivos, conservación de suelos, agroforestería, eficiencia en el uso de agua y adaptación al cambio climático. Las políticas públicas de países y regiones, como la Unión Europea, apoyan esta transición mediante subsidios ecológicos y programas agroambientales.



5

Modelado del sistema

Objetivos

El objetivo de este capítulo es introducir algunos tipos de modelo de sistema que pueden desarrollarse, como parte de la ingeniería de requerimientos y los procesos de diseño del sistema. Al estudiar este capítulo:

- comprenderá cómo usar los modelos gráficos para representar los sistemas de software;
- entenderá por qué se requieren diferentes tipos de modelo, así como las perspectivas fundamentales de contexto, interacción, estructura y comportamiento del modelado de sistemas;
- accederá a algunos de los tipos de diagrama en el Lenguaje de Modelado Unificado (UML) y conocerá cómo se utilizan dichos diagramas en el modelado del sistema;
- estará al tanto de las ideas que subyacen en la ingeniería dirigida por modelo, donde un sistema se genera automáticamente a partir de modelos estructurales y de comportamiento.

Contenido

El modelado de sistemas es el proceso para desarrollar modelos abstractos de un sistema, donde cada modelo presenta una visión o perspectiva diferente de dicho sistema. En general, el modelado de sistemas se ha convertido en un medio para representar el sistema usando algún tipo de notación gráfica, que ahora casi siempre se basa en notaciones en el Lenguaje de Modelado Unificado (UML). Sin embargo, también es posible desarrollar modelos formales (matemáticos) de un sistema, generalmente como una especificación detallada del sistema. En este capítulo se estudia el modelado gráfico utilizando el UML, y en el capítulo 12, el modelado formal.

Los modelos se usan durante el proceso de ingeniería de requerimientos para ayudar a derivar los requerimientos de un sistema, durante el proceso de diseño para describir el sistema a los ingenieros que implementan el sistema, y después de la implementación para documentar la estructura y la operación del sistema. Es posible desarrollar modelos tanto del sistema existente como del sistema a diseñar:

1. Los modelos del sistema existente se usan durante la ingeniería de requerimientos. Ayudan a aclarar lo que hace el sistema existente y pueden utilizarse como base para discutir sus fortalezas y debilidades. Posteriormente, conducen a los requerimientos para el nuevo sistema.
2. Los modelos del sistema nuevo se emplean durante la ingeniería de requerimientos para ayudar a explicar los requerimientos propuestos a otros participantes del sistema. Los ingenieros usan tales modelos para discutir las propuestas de diseño y documentar el sistema para la implementación. En un proceso de ingeniería dirigido por modelo, es posible generar una implementación de sistema completa o parcial a partir del modelo del sistema.

El aspecto más importante de un modelo del sistema es que deja fuera los detalles. Un modelo es una abstracción del sistema a estudiar, y no una representación alternativa de dicho sistema. De manera ideal, una representación de un sistema debe mantener toda la información sobre la entidad a representar. Una abstracción simplifica y recoge deliberadamente las características más destacadas. Por ejemplo, en el muy improbable caso de que este libro se entregue por capítulos en un periódico, la presentación sería una abstracción de los puntos clave del libro. Si se tradujera del inglés al italiano, sería una representación alternativa. La intención del traductor sería mantener toda la información como se presenta en inglés.

Desde diferentes perspectivas, usted puede desarrollar diferentes modelos para representar el sistema. Por ejemplo:

1. Una perspectiva externa, donde se modelen el contexto o entorno del sistema.
2. Una perspectiva de interacción, donde se modele la interacción entre un sistema y su entorno, o entre los componentes de un sistema.
3. Una perspectiva estructural, donde se modelen la organización de un sistema o la estructura de datos que procese el sistema.
4. Una perspectiva de comportamiento, donde se modele el comportamiento dinámico del sistema y cómo responde ante ciertos eventos.

Estas perspectivas tienen mucho en común con la visión 4 + 1 de arquitectura del sistema de Kruchten (Kruchten, 1995), la cual sugiere que la arquitectura y la organización de un sistema deben documentarse desde diferentes perspectivas. En el capítulo 6 se estudia este enfoque 4 + 1.

En este capítulo se usan diagramas definidos en UML (Booch *et al*., 2005; Rumbaugh *et al*., 2004), que se han convertido en un lenguaje de modelado estándar para modelado orientado a objetos. El UML tiene numerosos tipos de diagramas y, por lo tanto, soporta la creación de muchos diferentes tipos de modelo de sistema. Sin embargo, un estudio en 2007 (Erickson y Siau, 2007) mostró que la mayoría de los usuarios del UML consideraban que cinco tipos de diagrama podrían representar lo esencial de un sistema.

1. Diagramas de actividad, que muestran las actividades incluidas en un proceso o en el procesamiento de datos.
2. Diagramas de caso de uso, que exponen las interacciones entre un sistema y su entorno.
3. Diagramas de secuencias, que muestran las interacciones entre los actores y el sistema, y entre los componentes del sistema.
4. Diagramas de clase, que revelan las clases de objeto en el sistema y las asociaciones entre estas clases.
5. Diagramas de estado, que explican cómo reacciona el sistema frente a eventos internos y externos.

Como aquí no hay espacio para discutir todos los tipos de diagramas UML, el enfoque se centrará en cómo estos cinco tipos clave de diagramas se usan en el modelado del sistema.

Cuando desarrolle modelos de sistema, sea flexible en la forma en que use la notación gráfica. No siempre necesitará apegarse rigurosamente a los detalles de una notación. El detalle y el rigor de un modelo dependen de cómo lo use. Hay tres formas en que los modelos gráficos se emplean con frecuencia:

1. Como medio para facilitar la discusión sobre un sistema existente o propuesto.
2. Como una forma de documentar un sistema existente.
3. Como una descripción detallada del sistema que sirve para generar una implementación de sistema.

En el primer caso, el propósito del modelo es estimular la discusión entre los ingenieros de software que intervienen en el desarrollo del sistema. Los modelos pueden ser incompletos (siempre que cubran los puntos clave de la discusión) y utilizar de manera informal la notación de modelado. Así es como se utilizan en general los modelos en el llamado "modelado ágil" (Ambler y Jeffries, 2002). Cuando los modelos se usan como documentación, no tienen que estar completos, pues quizás usted sólo desee desarrollar modelos para algunas partes de un sistema. Sin embargo, estos modelos deben ser correctos: tienen que usar adecuadamente la notación y ser una descripción precisa del sistema.

El Lenguaje de Modelado Unificado

El Lenguaje de Modelado Unificado es un conjunto compuesto por 13 diferentes tipos de diagrama para modelar sistemas de software. Surgió del trabajo en la década de 1990 sobre el modelado orientado a objetos, cuando anotaciones similares, orientadas a objetos, se integraron para crear el UML. Una amplia revisión (UML 2) se finalizó en 2004. El UML es aceptado universalmente como el enfoque estándar al desarrollo de modelos de sistemas de software. Se han propuesto variantes más generales para el modelado de sistemas.

http://www.SoftwareEngineering-9.com/Web/UML/

En el tercer caso, en que los modelos se usan como parte de un proceso de desarrollo basado en modelo, los modelos de sistema deben ser completos y correctos. La razón para esto es que se usan como base para generar el código fuente del sistema. Por lo tanto, debe ser muy cuidadoso de no confundir símbolos equivalentes, como las flechas de palo y las de bloque, que tienen significados diferentes.

5.1 Modelos de contexto

En una primera etapa en 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. Tal vez decida que el apoyo automatizado para algunos procesos empresariales deba implementarse, pero otros deben ser procesos manuales o soportados por sistemas diferentes. Debe buscar posibles traslapes en la funcionalidad con los sistemas existentes y determinar dónde tiene que implementarse nueva funcionalidad. Estas decisiones deben hacerse oportunamente durante el proceso, para limitar los costos del sistema, así como el tiempo necesario para comprender los requerimientos y el diseño del sistema.

En algunos casos, la frontera entre un sistema y su entorno es relativamente clara. Por ejemplo, donde un sistema automático sustituye un sistema manual o computarizado, el entorno del nuevo sistema, por lo general, es el mismo que el entorno del sistema existente. En otros casos, existe más flexibilidad y usted es quien decide qué constituye la frontera entre el sistema y su entorno, durante el proceso de ingeniería de requerimientos.

Por ejemplo, imagine que desarrolla la especificación para el sistema de información de pacientes para atención a la salud mental. Este sistema intenta manejar la información sobre los pacientes que asisten a clínicas de salud mental y los tratamientos que les prescriben. Al desarrollar la especificación para este sistema, debe decidir si el sistema tiene que enfocarse exclusivamente en reunir información de las consultas (junto con otros sistemas para recopilar información personal acerca de los pacientes), o si también es necesario que recopile datos personales acerca del paciente. La ventaja de apoyarse en otros sistemas para la información del paciente es que evita duplicar datos. Sin embargo, la principal desventaja es que usar otros sistemas haría más lento el acceso a la información. Si estos sistemas no están disponibles, entonces no pueden usarse en MHC-PMS.

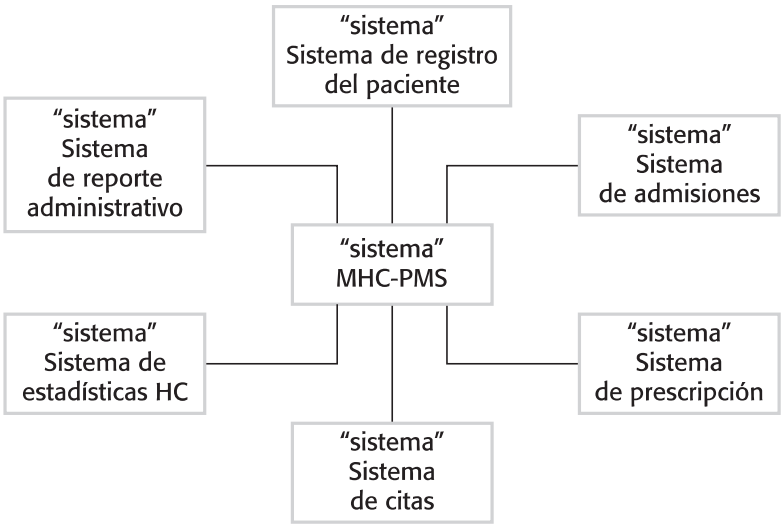


Figura 5.1 El contexto del MHC-PMS

La definición de frontera de un sistema no es un juicio libre de valor. Las preocupaciones sociales y organizacionales pueden significar que la posición de la frontera de un sistema se determine considerando factores no técnicos. Por ejemplo, una frontera de sistema puede colocarse deliberadamente, de modo que todo el proceso de análisis se realice en un sitio; puede elegirse de forma que sea innecesario consultar a un administrador particularmente difícil; puede situarse de manera que el costo del sistema aumente y la división de desarrollo del sistema deba, por lo tanto, expandirse al diseño y la implementación del sistema.

Una vez tomadas algunas decisiones sobre las fronteras del sistema, parte de la actividad de análisis es la definición de dicho contexto y las dependencias que un sistema tiene con su entorno. Normalmente, producir un modelo arquitectónico simple es el primer paso en esta actividad.

La figura 5.1 es un modelo de contexto simple que muestra el sistema de información del paciente y otros sistemas en su entorno. A partir de la figura 5.1, se observa que el MHC-PMS está conectado con un sistema de citas y un sistema más general de registro de pacientes, con el cual comparte datos. El sistema también está conectado a sistemas para manejo de reportes y asignación de camas de hospital, y un sistema de estadísticas que recopila información para la investigación. Finalmente, utiliza un sistema de prescripción que elabora recetas para la medicación de los pacientes.

Los modelos de contexto, por lo general, muestran que el entorno incluye varios sistemas automatizados. Sin embargo, no presentan los tipos de relaciones entre los sistemas en el entorno y el sistema que se especifica. Los sistemas externos generan datos para el sistema o consumen datos del sistema. Pueden compartir datos con el sistema, conectarse directamente, a través de una red, o no conectarse en absoluto. Pueden estar físicamente juntos o ubicados en edificios separados. Todas estas relaciones llegan a afectar los requerimientos y el diseño del sistema a definir, por lo que deben tomarse en cuenta.

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.

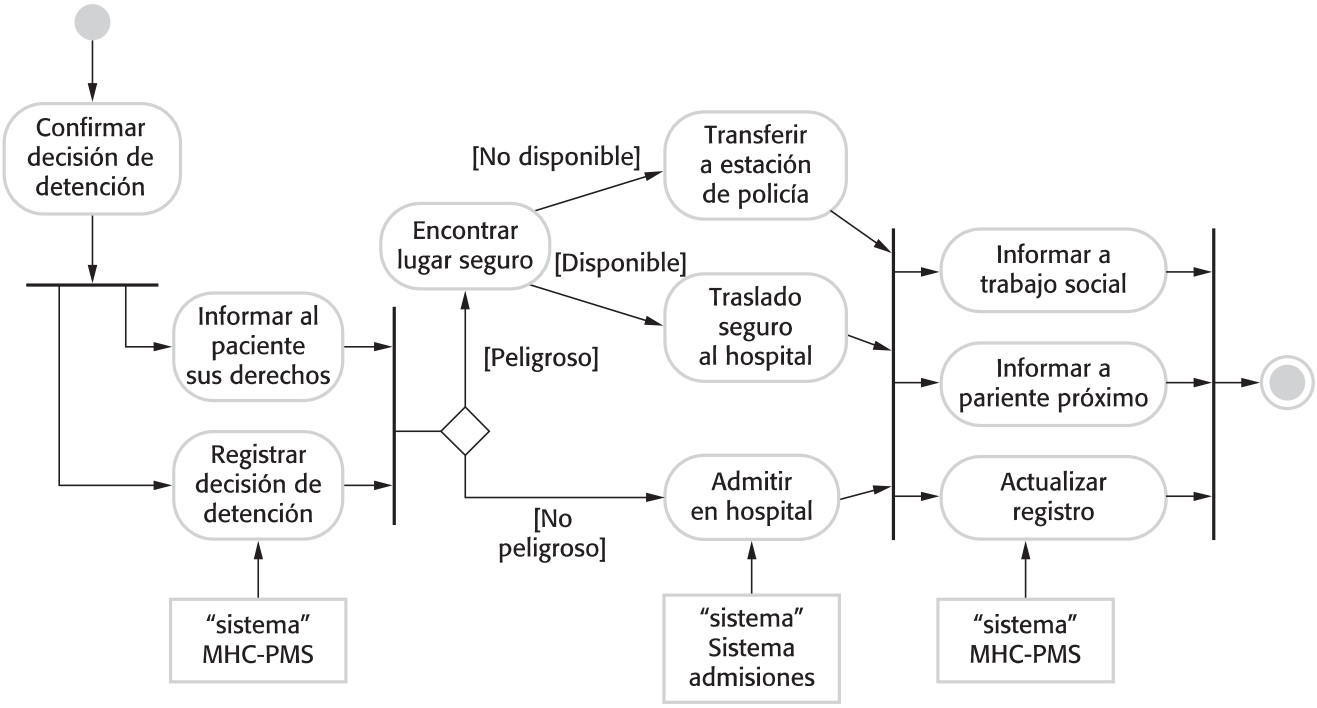


Figura 5.2 Modelo del proceso de detención involuntaria

La figura 5.2 es un modelo de un importante proceso de sistema que muestra los procesos en que se utiliza el MHC-PMS. En ocasiones, los pacientes que sufren de problemas de salud mental son un riesgo para otros o para sí mismos. Por ello, es posible que en un hospital deban mantenerse contra su voluntad para que se les suministre el tratamiento. Tal detención está sujeta a estrictas protecciones legales, por ejemplo, la decisión de detener a un paciente tiene que revisarse con regularidad, para que no se detenga a la persona indefinidamente sin una buena razón. Una de las funciones del MHC-PMS es garantizar que se implementen dichas protecciones.

La figura 5.2 es un diagrama de actividad UML. Los diagramas de actividad intentan mostrar las actividades que incluyen un proceso de sistema, así como el flujo de control de una actividad a otra. El inicio de un proceso se indica con un círculo lleno; el fin, mediante un círculo lleno dentro de otro círculo. Los rectángulos con esquinas redondeadas representan actividades, esto es, los subprocesos específicos que hay que realizar. Puede incluir objetos en los gráficos de actividad. En la figura 5.2 se muestran los sistemas que sirven para apoyar diferentes procesos. Se indicó que éstos son sistemas separados al usar la característica de estereotipo UML.

En un diagrama de actividad UML, las flechas representan el flujo de trabajo de una actividad a otra. Una barra sólida se emplea para indicar coordinación de actividades. Cuando el flujo de más de una actividad se dirige a una barra sólida, entonces todas esas actividades deben completarse antes del posible avance. Cuando el flujo de una barra sólida conduzca a algunas actividades, éstas pueden ejecutarse en forma paralela. Por consiguiente, en la figura 5.2, las actividades para informar a trabajo social y al familiar cercano del paciente, así como para actualizar el registro de detención, pueden ser concurrentes.

Las flechas pueden anotarse con guardas que indiquen la condición al tomar dicho flujo. En la figura 5.2 se observan guardas que muestran los flujos para pacientes que son

un riesgo para la sociedad y quienes no lo son. Los pacientes peligrosos deben mantenerse en una instalación segura. No obstante, los pacientes suicidas que, por lo tanto, representan un riesgo para sí mismos, se detendrían en un pabellón hospitalario adecuado.

5.2 Modelos de interacción

Todos los sistemas incluyen interacciones de algún tipo. Éstas son interacciones del usuario, que implican entradas y salidas del usuario; interacciones entre el sistema a desarrollar y otros sistemas; o interacciones entre los componentes del sistema. El modelado de interacción del usuario es importante, pues ayuda a identificar los requerimientos del usuario. El modelado de la interacción sistema a sistema destaca los problemas de comunicación que se lleguen a presentar. El modelado de interacción de componentes ayuda a entender si es probable que una estructura de un sistema propuesto obtenga el rendimiento y la confiabilidad requeridos por el sistema.

En esta sección se cubren dos enfoques relacionados con el modelado de interacción:

1. Modelado de caso de uso, que se utiliza principalmente para modelar interacciones entre un sistema y actores externos (usuarios u otros sistemas).
2. Diagramas de secuencia, que se emplean para modelar interacciones entre componentes del sistema, aunque también pueden incluirse agentes externos.

Los modelos de caso de uso y los diagramas de secuencia presentan la interacción a diferentes niveles de detalle y, por lo tanto, es posible utilizarlos juntos. Los detalles de las interacciones que hay en un caso de uso de alto nivel se documentan en un diagrama de secuencia. El UML también incluye diagramas de comunicación usados para modelar interacciones. Aquí no se analiza esto, ya que se trata de representaciones alternativas de gráficos de secuencia. De hecho, algunas herramientas pueden generar un diagrama de comunicación a partir de un diagrama de secuencia.

5.2.1 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. La figura 5.3 presenta un caso de uso del MHC-PMS que implica la tarea de subir datos desde el MHC-PMS hasta un sistema más general de registro de pacientes. Este sistema más general mantiene un resumen de datos sobre el paciente, en vez de los datos sobre cada consulta, que se registran en el MHC-PMS.

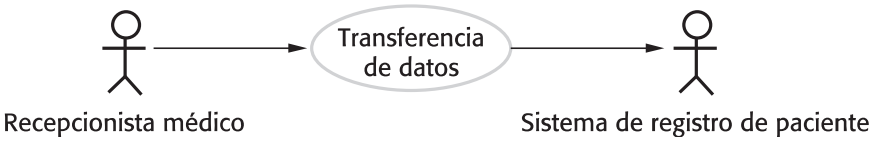


Figura 5.3 Caso de uso de transferencia de datos

Observe que en este caso de uso hay dos actores: el operador que transfiere los datos y el sistema de registro de pacientes. La notación con figura humana se desarrolló originalmente para cubrir la interacción entre individuos, pero también se usa ahora para representar otros sistemas externos y el hardware. De manera formal, los diagramas de caso de uso deben emplear líneas sin flechas; las flechas en el UML indican la dirección del flujo de mensajes. Evidentemente, en un caso de uso los mensajes pasan en ambas direcciones. Sin embargo, las flechas en la figura 5.3 se usan de manera informal para indicar que la recepcionista médica inicia la transacción y los datos se transfieren al sistema de registro de pacientes.

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. La figura 5.4 ilustra una descripción tabular del caso de uso "transferencia de datos".

Como vimos en el capítulo 4, los diagramas de caso de uso compuestos indican un número de casos de uso diferentes. En ocasiones, se incluyen todas las interacciones posibles con un sistema en un solo diagrama de caso de uso compuesto. Sin embargo, esto quizá sea imposible debido a la cantidad de casos de uso. En tales situaciones, puede desarrollar varios diagramas, cada uno de los cuales exponga casos de uso relacionados. Por ejemplo, la figura 5.5 presenta todos los casos de uso en el MHC-PMS, en los cuales interviene el actor "recepcionista médico".

Figura 5.4 Descripción tabular del caso de uso "transferencia de datos"

MHC-PMS: Transferencia de datos	
Actores	Recepcionista médico, sistema de registros de paciente (PRS).
Descripción	Un recepcionista puede transferir datos del MHC-PMS a una base de datos general de registro de pacientes, mantenida por una autoridad sanitaria. La información transferida puede ser información personal actualizada (dirección, número telefónico, etc.) o un resumen del diagnóstico y tratamiento del paciente.
Datos	Información personal del paciente, resumen de tratamiento.
Estímulo	Comando de usuario emitido por recepcionista médico.
Respuesta	Confirmación de que el PRS se actualizó.
Comentarios	El recepcionista debe tener permisos de seguridad adecuados para acceder a la información del paciente y al PRS.

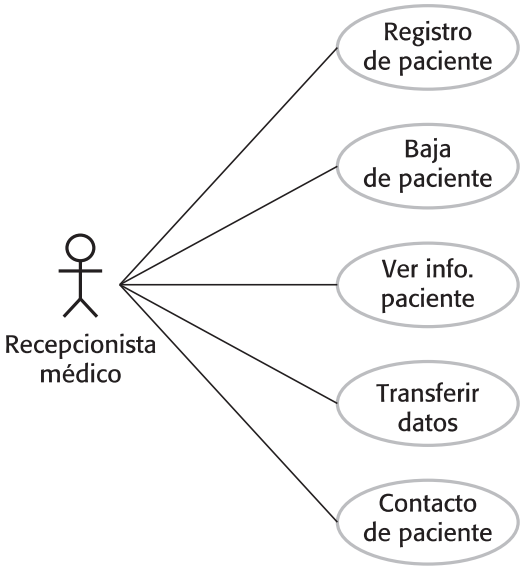


Figura 5.5 Casos de uso que involucran el papel "recepcionista médico"

5.2.2 Diagramas de secuencia

Los diagramas de secuencia en el UML se usan principalmente para modelar las interacciones entre los actores y los objetos en un sistema, así como las interacciones entre los objetos en sí. El UML tiene una amplia sintaxis para diagramas de secuencia, lo cual permite muchos tipos diferentes de interacción a modelar. Como aquí no hay espacio para cubrir todas las posibilidades, sólo nos enfocaremos en lo básico de este tipo de diagrama.

Como sugiere el nombre, un diagrama de secuencia muestra la sucesión de interacciones que ocurre durante un caso de uso particular o una instancia de caso de uso. La figura 5.6 es un ejemplo de un diagrama de secuencia que ilustra los fundamentos de la notación. Estos modelos de diagrama incluyen las interacciones en el caso de uso "ver información de paciente", donde un recepcionista médico puede conocer la información de algún paciente.

Los objetos y actores que intervienen se mencionan a lo largo de la parte superior del diagrama, con una línea punteada que se dibuja verticalmente a partir de éstos. Las interacciones entre los objetos se indican con flechas dirigidas. El rectángulo sobre las líneas punteadas indica la línea de vida del objeto tratado (es decir, el tiempo que la instancia del objeto está involucrada en la computación). La secuencia de interacciones se lee de arriba abajo. Las anotaciones sobre las flechas señalan las llamadas a los objetos, sus parámetros y los valores que regresan. En este ejemplo, también se muestra la notación empleada para exponer alternativas. Un recuadro marcado con "alt" se usa con las condiciones indicadas entre corchetes.

La figura 5.6 se lee del siguiente modo:

1. El recepcionista médico activa el método ViewInfo (ver información) en una instancia P de la clase de objeto PatientInfo, y suministra el identificador del paciente, PID. P es un objeto de interfaz de usuario, que se despliega como un formato que muestra la información del paciente.
2. La instancia P llama a la base de datos para regresar la información requerida, y suministra el identificador del recepcionista para permitir la verificación de seguridad (en esta etapa no se preocupe de dónde proviene este UID).

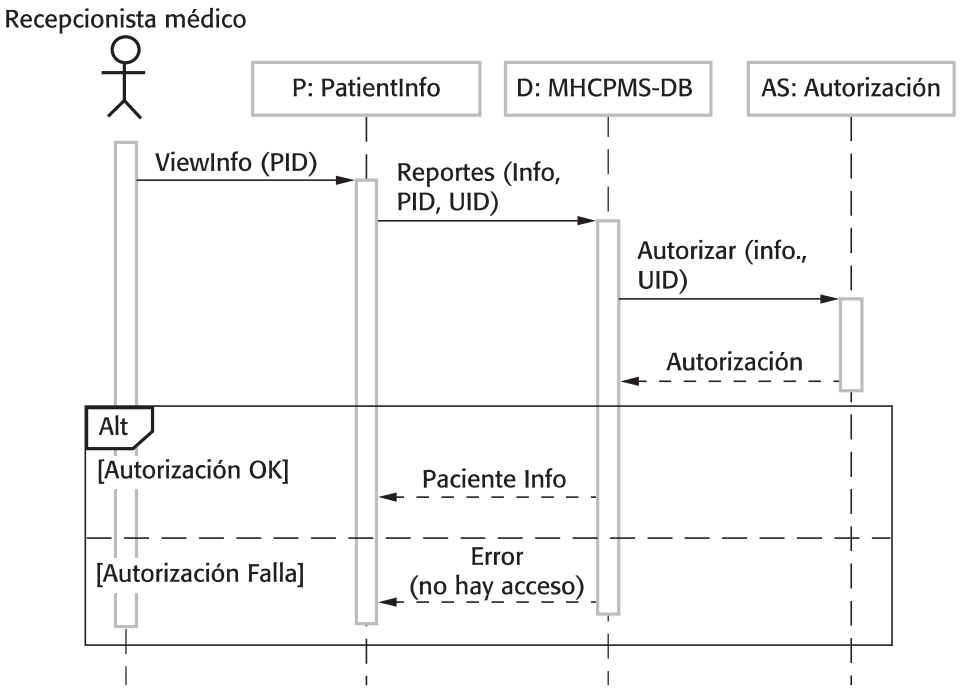


Figura 5.6 Diagrama de secuencia para "ver información del paciente"

3. La base de datos comprueba, mediante un sistema de autorización, que el usuario esté autorizado para tal acción.
4. Si está autorizado, se regresa la información del paciente y se llena un formato en la pantalla del usuario. Si la autorización falla, entonces se regresa un mensaje de error.

La figura 5.7 es un segundo ejemplo de un diagrama de secuencia del mismo sistema que ilustra dos características adicionales. Se trata de la comunicación directa entre los actores en el sistema y la creación de objetos como parte de una secuencia de operaciones. En este ejemplo, un objeto del tipo Summary (resumen) se crea para contener los datos del resumen que deben subirse al PRS (*patient record system*, es decir, el sistema de registro de paciente). Este diagrama se lee de la siguiente manera:

1. El recepcionista inicia sesión (log) en el PRS.
2. Hay dos opciones disponibles. Las opciones permiten la transferencia directa de información actualizada del paciente al PRS, y la transferencia de datos del resumen de salud del MHC-PMS al PRS.
3. En cada caso, se verifican los permisos del recepcionista usando el sistema de autorización.
4. La información personal se transfiere directamente del objeto de interfaz del usuario al PRS. De manera alternativa, es posible crear un registro del resumen de la base de datos y, luego, transferir dicho registro.
5. Al completar la transferencia, el PRS emite un mensaje de estatus y el usuario termina la sesión (log off).

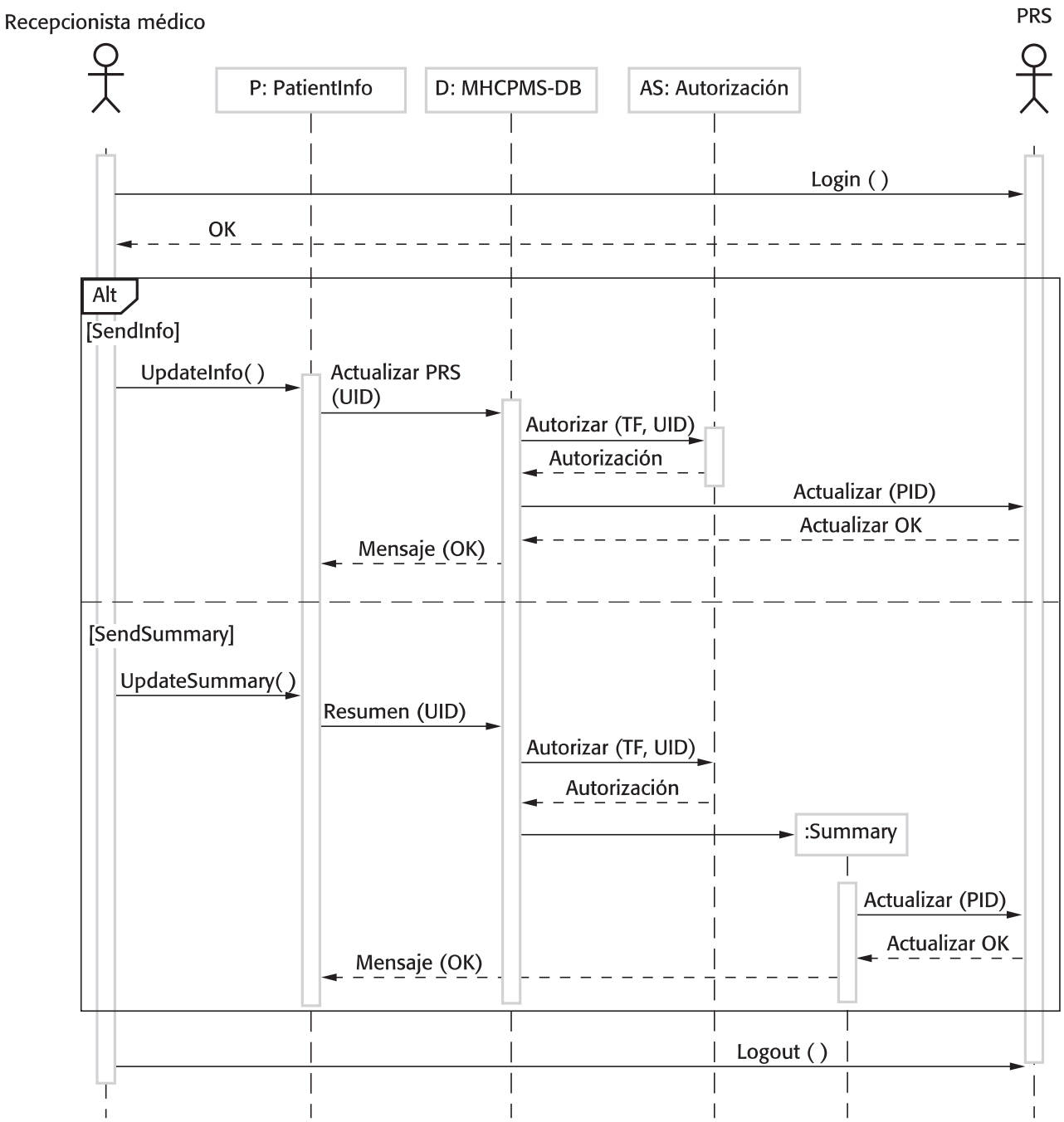


Figura 5.7 Diagrama de secuencia para transferir datos

A menos que use diagramas de secuencia para generación de código o documentación detallada, en dichos diagramas no tiene que incluir todas las interacciones. Si desarrolla modelos iniciales de sistema en el proceso de desarrollo para apoyar la ingeniería de requerimientos y el diseño de alto nivel, habrá muchas interacciones que dependan de decisiones de implementación. Por ejemplo, en la figura 5.7, la decisión sobre cómo conseguir el identificador del usuario para comprobar la autorización podría demorarse. En una implementación, esto implicaría la interacción con un objeto User (usuario), pero esto no es importante en esta etapa y, por lo tanto, no necesita incluirse en el diagrama de secuencia.

Análisis de requerimientos orientado a objetos

En el análisis de requerimientos orientado a objetos, se modelan entidades del mundo real usando clases de objetos. Usted puede crear diferentes tipos de modelos de objetos, que muestren cómo se relacionan mutuamente las clases de objetos, cómo se agregan objetos para formar otros objetos, cómo interactúan los objetos entre sí, etcétera. Cada uno de éstos presenta información única acerca del sistema que se especifica.

http://www.SoftwareEngineering-9.com/Web/OORA/

5.3 Modelos estructurales

Los modelos estructurales de software muestran la organización de un sistema, en términos de los componentes que constituyen dicho sistema y sus relaciones. Los modelos estructurales son modelos estáticos, que muestran la estructura del diseño del sistema, o modelos dinámicos, que revelan la organización del sistema cuando se ejecuta. No son lo mismo: la organización dinámica de un sistema como un conjunto de hilos en interacción tiende a ser muy diferente de un modelo estático de componentes del sistema.

Los modelos estructurales de un sistema se crean cuando se discute y diseña la arquitectura del sistema. El diseño arquitectónico es un tema particularmente importante en la ingeniería de software, y los diagramas UML de componente, de paquete y de implementación se utilizan cuando se presentan modelos arquitectónicos. En los capítulos 6, 18 y 19 se cubren diferentes aspectos de la arquitectura de software y del modelado arquitectónico. Esta sección se enfoca en el uso de diagramas de clase para modelar la estructura estática de las clases de objetos, en un sistema de software.

5.3.1 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 dicha clases. En consecuencia, cada clase puede tener algún conocimiento de esta clase asociada.

Cuando se desarrollan modelos durante las primeras etapas del proceso de ingeniería de software, los objetos representan algo en el mundo real, como un paciente, una receta, un médico, etcétera. Conforme se desarrolla una implementación, por lo general se necesitan definir los objetos de implementación adicionales que se usan para dar la funcionalidad requerida del sistema. Aquí, el enfoque está sobre el modelado de objetos del mundo real, como parte de los requerimientos o los primeros procesos de diseño del software.

Los diagramas de clase en el UML pueden expresarse con diferentes niveles de detalle. Cuando se desarrolla un modelo, la primera etapa con frecuencia implica buscar en el mundo, identificar los objetos esenciales y representarlos como clases. La forma más sencilla de hacer esto es escribir el nombre de la clase en un recuadro. También puede anotar la existencia de una asociación dibujando simplemente una línea entre las clases.

Figura 5.8 Clases y asociación UML

Por ejemplo, la figura 5.8 es un diagrama de clase simple que muestra dos clases: Patient (paciente) y Patient Record (registro del paciente), con una asociación entre ellos.

En la figura 5.8 se ilustra una característica más de los diagramas de clase: la habilidad para mostrar cuántos objetos intervienen en la asociación. En este ejemplo, cada extremo de la asociación se registra con un 1, lo cual significa que hay una relación 1:1 entre objetos de dichas clases. Esto es, cada paciente tiene exactamente un registro, y cada registro conserva información precisa del paciente. En los últimos ejemplos se observa que son posibles otras multiplicidades. Se define un número exacto de objetos que están implicados, o bien, con el uso de un asterisco (*), como se muestra en la figura 5.9, que hay un número indefinido de objetos en la asociación.

La figura 5.9 desarrolla este tipo de diagrama de clase para mostrar que los objetos de la clase "paciente" también intervienen en relaciones con varias otras clases. En este ejemplo, se observa que es posible nombrar las asociaciones para dar al lector un indicio del tipo de relación que existe. Asimismo, el UML permite especificar el papel de los objetos que participan en la asociación.

En este nivel de detalle, los diagramas de clase parecen modelos semánticos de datos. Los modelos semánticos de datos se usan en el diseño de bases de datos. Muestran las entidades de datos, sus atributos asociados y las relaciones entre dichas entidades. Este enfoque para modelar fue propuesto por primera vez por Chen (1976), a mediados de la década de 1970; desde entonces, se han desarrollado diversas variantes (Codd, 1979; Hammer y McLeod, 1981; Hull y King, 1987), todas con la misma forma básica.

El UML no incluye una notación específica para este modelado de bases de datos, ya que supone un proceso de desarrollo orientado a objetos, así como modelos de datos que usan objetos y sus relaciones. Sin embargo, es posible usar el UML para representar un modelo semántico de datos. En un modelo semántico de datos, piense en entidades como

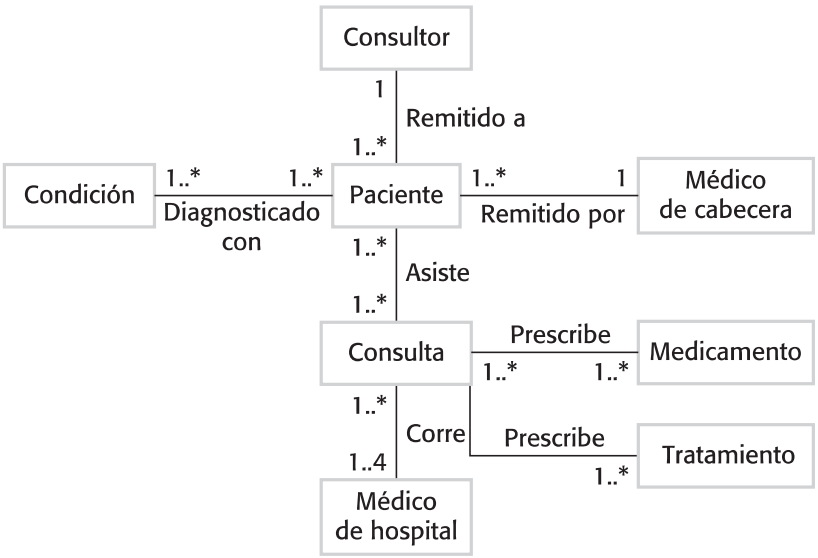


Figura 5.9 Clases y asociaciones en el MHC-PMS

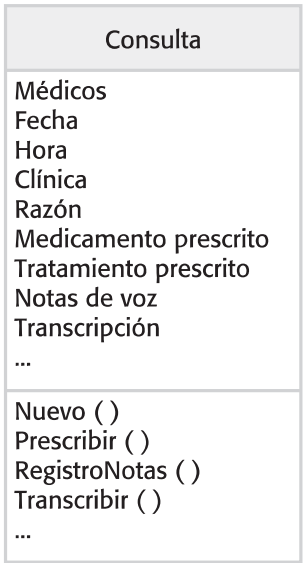


Figura 5.10 La clase de consulta

clases de objeto simplificadas (no tienen operaciones), atributos como atributos de clase de objeto y relaciones como nombres de asociaciones entre clases de objeto.

Cuando se muestran las asociaciones entre clases, es conveniente representar dichas clases en la forma más sencilla posible. Para definirlas con más detalle, agregue información sobre sus atributos (las características de un objeto) y operaciones (aquello que se puede solicitar de un objeto). Por ejemplo, un objeto Patient tendrá el atributo Address (dirección) y puede incluir una operación llamada ChangeAddress (cambiar dirección), que se llama cuando un paciente manifiesta que se mudó de una dirección a otra. En el UML, los atributos y las operaciones se muestran al extender el rectángulo simple que representa una clase. Esto se ilustra en la figura 5.10, donde:

1. El nombre de la clase de objeto está en la sección superior.
2. Los atributos de clase están en la sección media. Esto debe incluir los nombres del atributo y, opcionalmente, sus tipos.
3. Las operaciones (llamadas métodos en Java y en otros lenguajes de programación OO) asociadas con la clase de objeto están en la sección inferior del rectángulo.

La figura 5.10 expone posibles atributos y operaciones sobre la clase Consulta (Consultation). En este ejemplo, se supone que los médicos registran notas de voz que se transcriben más tarde para registrar detalles de la consulta. Al prescribir fármacos, el médico debe usar el método Prescribir (Prescribe) para generar una receta electrónica.

5.3.2 Generalización

La generalización es una técnica cotidiana que se usa para gestionar la complejidad. En vez de aprender las características detalladas de cada entidad que se experimenta, dichas entidades se colocan en clases más generales (animales, automóviles, casas,

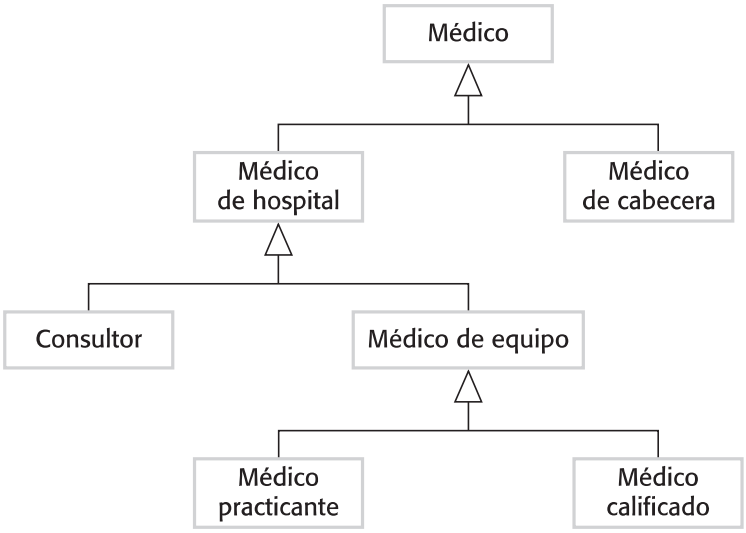


Figura 5.11 Jerarquía de generalización

etcétera) y se aprenden las características de dichas clases. Esto permite deducir que diferentes miembros de estas clases tienen algunas características comunes (por ejemplo, las ardillas y ratas son roedores). Es posible hacer enunciados generales que se apliquen a todos los miembros de la clase (por ejemplo, todos los roedores tienen dientes para roer).

En el modelado de sistemas, con frecuencia es útil examinar las clases en un sistema, con la finalidad de ver si hay ámbito para la generalización. Esto significa que la información común se mantendrá solamente en un lugar. Ésta es una buena práctica de diseño, pues significa que, si se proponen cambios, entonces no se tiene que buscar en todas las clases en el sistema, para observar si se ven afectadas por el cambio. En los lenguajes orientados a objetos, como Java, la generalización se implementa usando los mecanismos de herencia de clase construidos en el lenguaje.

El UML tiene un tipo específico de asociación para denotar la generalización, como se ilustra en la figura 5.11. La generalización se muestra como una flecha que apunta hacia la clase más general. Esto indica que los médicos de cabecera y los médicos de hospital pueden generalizarse como médicos, y que hay tres tipos de médicos de hospital: quienes se graduaron recientemente de la escuela de medicina y tienen que ser supervisados (médicos practicantes); quienes trabajan sin supervisión como parte de un equipo de consultores (médicos registrados); y los consultores, que son médicos experimentados con plenas responsabilidades en la toma de decisiones.

En una generalización, los atributos y las operaciones asociados con las clases de nivel superior también se asocian con las clases de nivel inferior. En esencia, las clases de nivel inferior son subclases que heredan los atributos y las operaciones de sus superclases. Entonces dichas clases de nivel inferior agregan atributos y operaciones más específicos. Por ejemplo, todos los médicos tienen un nombre y número telefónico; todos los médicos de hospital tienen un número de personal y un departamento, pero los médicos de cabecera no tienen tales atributos, pues trabajan de manera independiente. Sin embargo, sí tienen un nombre de consultorio y dirección. Esto se ilustra en la figura 5.12, que muestra parte de la jerarquía de generalización que se extendió con atributos de clase. Las operaciones asociadas con la clase “médico” buscan registrar y dar de baja al médico con el MHC-PMS.

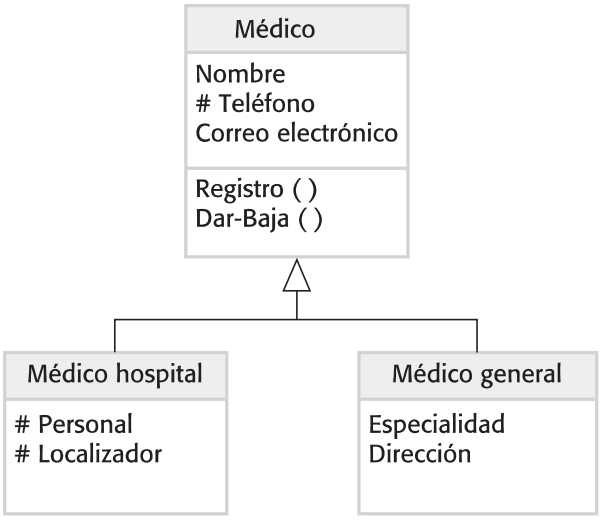


Figura 5.12 Jerarquía de generalización con detalles agregados

5.3.3 Agregación

Los objetos en el mundo real con frecuencia están compuestos por diferentes partes. Un paquete de estudio para un curso, por ejemplo, estaría compuesto por libro, diapositivas de PowerPoint, exámenes y recomendaciones para lecturas posteriores. En ocasiones, en un modelo de sistema, usted necesita ilustrar esto. El UML proporciona un tipo especial de asociación entre clases llamado agregación, que significa que un objeto (el todo) se compone de otros objetos (las partes). Para mostrarlo, se usa un trazo en forma de diamante, junto con la clase que representa el todo. Esto se ilustra en la figura 5.13, que indica que un registro de paciente es una composición de Paciente (Patient) y un número indefinido de Consulta (Consultations).

5.4 Modelos de comportamiento

Los modelos de comportamiento son modelos dinámicos del sistema conforme se ejecutan. En ellos se muestra lo que sucede o lo que se supone que pasa cuando un sistema responde ante un estímulo de su entorno. Tales estímulos son de dos tipos:

1. *Datos* Algunos datos que llegan se procesan por el sistema.
2. *Eventos* Algunos eventos activan el procesamiento del sistema. Los eventos pueden tener datos asociados, pero esto no es siempre el caso.

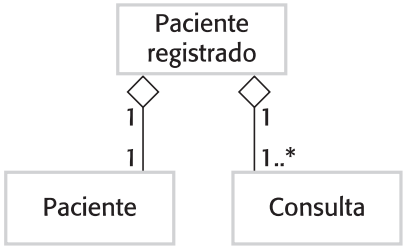


Figura 5.13 La asociación agregación

Diagramas de flujo de datos

Los diagramas de flujo de datos (DFD) son modelos de sistema que presentan una perspectiva funcional, donde cada transformación constituye una sola función o un solo proceso. Los DFD se usan para mostrar cómo fluyen los datos a través de una secuencia de pasos del procesamiento. Por ejemplo, un paso del procesamiento sería el filtrado de registros duplicados en una base de datos de clientes. Los datos se transforman en cada paso antes de moverse hacia la siguiente etapa. Dichos pasos o transformaciones del procesamiento representan procesos o funciones de software, en los cuales los diagramas de flujo de datos se usan para documentar un diseño de software.

http://www.SoftwareEngineering-9.com/Web/DFDs

Muchos sistemas empresariales son sistemas de procesamiento de datos que se activan principalmente por datos. Son controlados por la entrada de datos al sistema con relativamente poco procesamiento externo de eventos. Su procesamiento incluye una secuencia de acciones sobre dichos datos y la generación de una salida. Por ejemplo, un sistema de facturación telefónica aceptará información de las llamadas hechas por un cliente, calculará los costos de dichas llamadas y generará una factura para enviarla a dicho cliente. En contraste, los sistemas de tiempo real muchas veces están dirigidos por un evento con procesamiento de datos mínimo. Por ejemplo, un sistema de conmutación telefónico terrestre responde a eventos como "receptor ocupado" al generar un tono de dial, o al presionar las teclas de un teléfono para la captura del número telefónico, etcétera.

5.4.1 Modelado dirigido por datos

Los modelos dirigidos por datos muestran la secuencia de acciones involucradas en el procesamiento de datos de entrada, así como la generación de una salida asociada. Son particularmente útiles durante el análisis de requerimientos, pues sirven para mostrar procesamiento "extremo a extremo" en un sistema. Esto es, exhiben toda la secuencia de acciones que ocurren desde una entrada a procesar hasta la salida correspondiente, que es la respuesta del sistema.

Los modelos dirigidos por datos están entre los primeros modelos gráficos de software. En la década de 1970, los métodos estructurados como el análisis estructurado de DeMarco (DeMarco, 1978) introdujeron los diagramas de flujo de datos (DFD), como una forma de ilustrar los pasos del procesamiento en un sistema. Los modelos de flujo de datos son útiles porque el hecho de rastrear y documentar cómo los datos asociados con un proceso particular se mueven a través del sistema ayuda a los analistas y diseñadores a entender lo que sucede. Los diagramas de flujo de datos son simples e intuitivos y, por lo general, es posible explicarlos a los usuarios potenciales del sistema, quienes después pueden participar en la validación del modelo.

El UML no soporta diagramas de flujo de datos, puesto que originalmente se propusieron y usaron para modelar el procesamiento de datos. La razón para esto es que los DFD se enfocan en funciones del sistema y no reconocen objetos del sistema. Sin embargo, como los sistemas dirigidos por datos son tan comunes en los negocios, UML 2.0 introdujo diagramas de actividad, que son similares a los diagramas de flujo de datos. Por ejemplo, la figura 5.14 indica la cadena de procesamiento involucrada en el software de la bomba de insulina. En este diagrama, se observan los pasos de procesamiento (representados como actividades) y los datos que fluyen entre dichos pasos (representados como objetos).

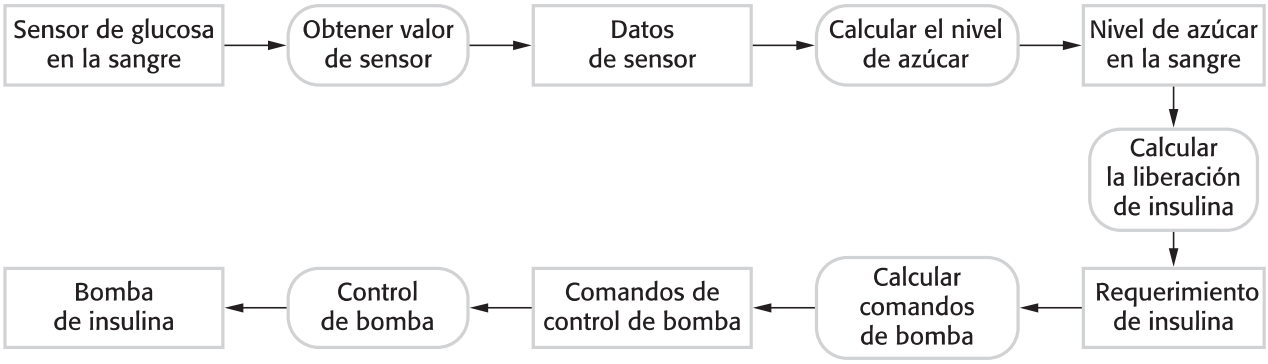


Figura 5.14 Modelo de actividad de la operación de la bomba de insulina

Una forma alternativa de mostrar la secuencia de procesamiento en un sistema es usar diagramas de secuencia UML. Ya se vio cómo se utilizan para modelar interacción pero, si los utiliza para que dichos mensajes sólo se envíen de izquierda a derecha, luego muestran el procesamiento secuencial de datos en el sistema. La figura 5.15 ilustra esto, con un modelo de secuencia del procesamiento de un pedido y envío a un proveedor. Los modelos de secuencia destacan los objetos en un sistema, mientras que los diagramas de flujo de datos resaltan las funciones. El diagrama de flujo de datos equivalente para la orden de procesamiento se incluye en las páginas Web del libro.

5.4.2 Modelado dirigido por un evento

El modelado dirigido por un evento muestra cómo responde un sistema a eventos externos e internos. Se basa en la suposición de que un sistema tiene un número finito de estados y que los eventos (estímulos) pueden causar una transición de un estado a otro. Por ejemplo, un sistema que controla una válvula puede moverse de un estado de "válvula abierta" a un estado de "válvula cerrada", cuando recibe un comando operador (el estímulo). Esta visión de un sistema es adecuado particularmente para sistema en tiempo real. El modelado basado en eventos se introdujo en los métodos de diseño en tiempo real, como los propuestos por Ward y Mellor (1985) y Harel (1987, 1988).

El UML soporta modelado basado en eventos usando diagramas de estado, que se fundamentaron en gráficos de estado (Harel, 1987, 1988). Los diagramas de estado muestran estados y eventos del sistema que causan transiciones de un estado a otro. No exponen el

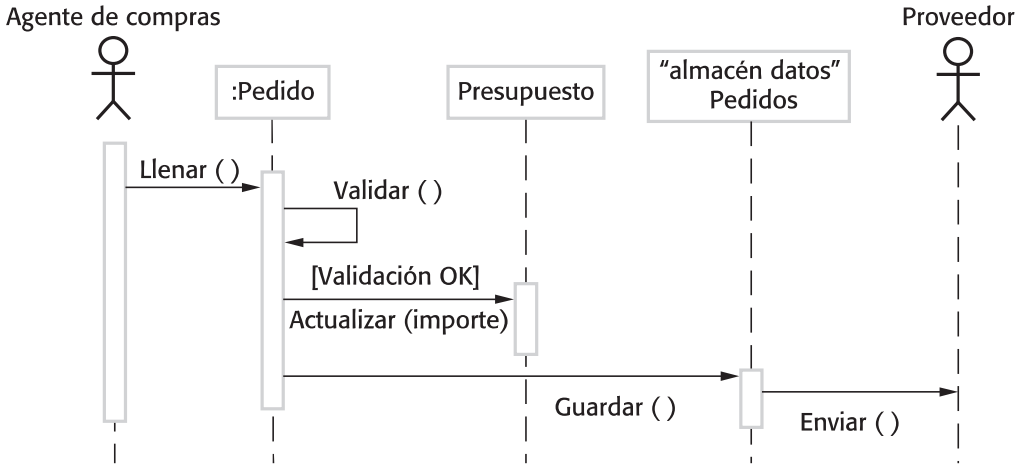


Figura 5.15 Orden de procesamiento

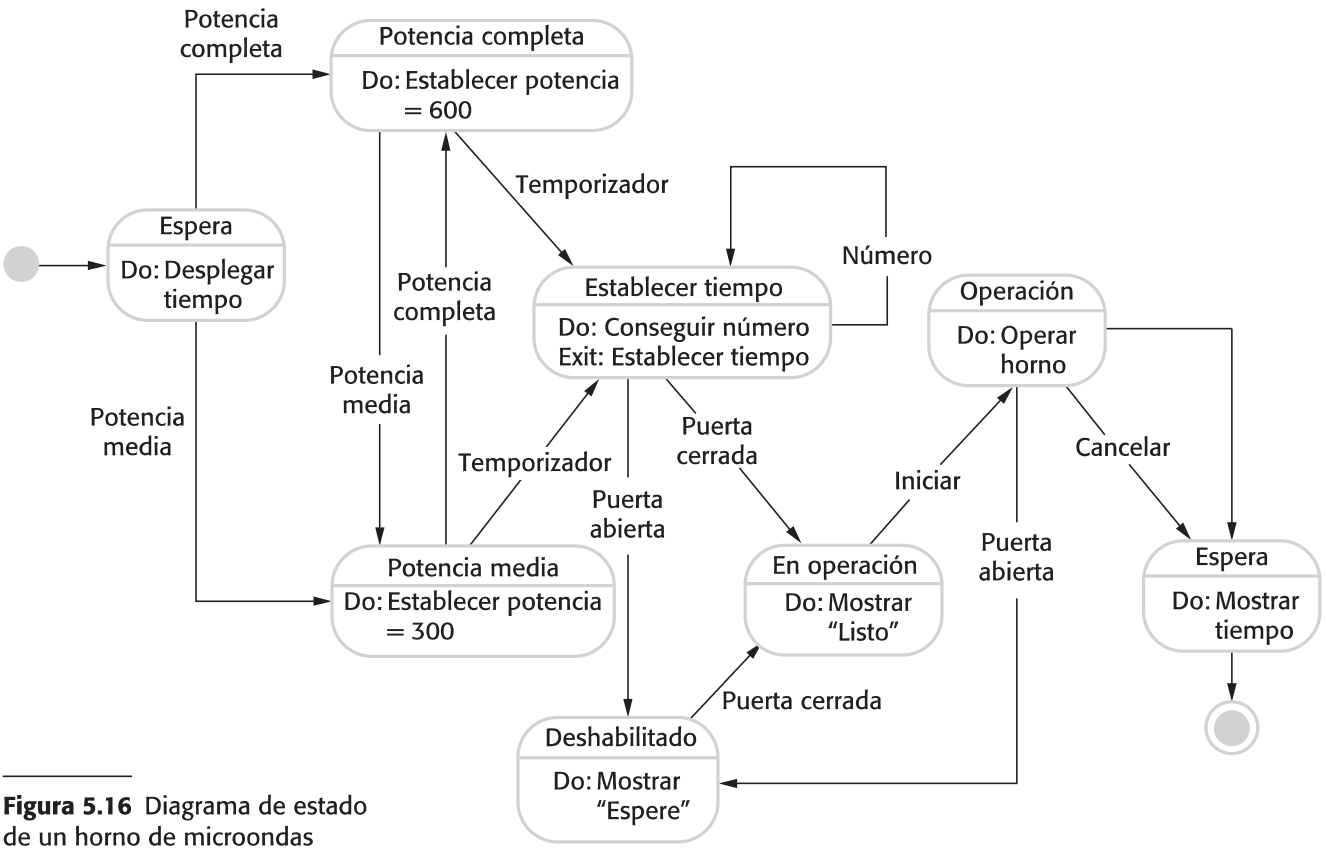


Figura 5.16 Diagrama de estado de un horno de microondas

flujo de datos dentro del sistema, pero suelen incluir información adicional acerca de los cálculos realizados en cada estado.

Se usa un ejemplo de software de control para un horno de microondas muy sencillo, que ilustra el modelado dirigido por un evento. Los hornos de microondas reales son mucho más complejos que este sistema, pero el sistema simplificado es más fácil de entender. Este microondas sencillo tiene un interruptor para seleccionar potencia completa o media, un teclado numérico para ingresar el tiempo de cocción, un botón de iniciar/detener y una pantalla alfanumérica.

Se supone que la secuencia de acciones al usar el horno de microondas es:

1. Seleccionar el nivel de potencia (ya sea media o completa)
2. Ingresar el tiempo de cocción con el teclado numérico.
3. Presionar "Iniciar", y la comida se cocina durante el tiempo dado.

Por razones de seguridad, el horno no opera cuando la puerta esté abierta y, al completar la cocción, se escuchará un timbre. El horno tiene una pantalla alfanumérica muy sencilla que se usa para mostrar varios avisos de alerta y mensajes de advertencia.

En los diagramas de estado UML, los rectángulos redondeados representan estados del sistema. Pueden incluir una breve descripción (después de "do") de las acciones que se tomarán en dicho estado. Las flechas etiquetadas representan estímulos que fuerzan una transición de un estado a otro. Puede indicar los estados inicial y final usando círculos rellenos, como en los diagramas de actividad.

A partir de la figura 5.16 se observa que el sistema empieza en un estado de espera e, inicialmente, responde al botón de potencia completa o al botón de potencia media. Los

Guía rápida para elegir el diagrama UML correcto

Necesidad	Diagrama recomendado	Qué muestra	Cuando usarlo
Ver quién interactúa con el sistema y qué funciones principales hay	Casos de uso	Actores externos + funcionalidades	Etapa inicial de análisis de requerimientos
Describir la estructura y relaciones entre las entidades	Clases	Clases, atributos, métodos, relaciones (herencia, agregación, etc.)	Diseño de la base de datos y lógica de negocio
Explicar el flujo paso a paso de un proceso	Actividades	Actividades, decisiones, bifurcaciones y concurrencia	Modelar procesos o flujos de trabajo
Mostrar cómo cambia un objeto según eventos	Estados	Estados posibles y transiciones	Sistemas con ciclos de vida claros (pedidos, tickets, etc.)
Visualizar el orden de interacción entre componentes	Secuencia	Participantes + mensajes enviados en el tiempo	Modelar comunicaciones en un escenario concreto
Representar el sistema y su entorno	Contexto	Sistema como caja negra + entidades externas	Cuando quieres ver límites y relaciones externas
Mostrar jerarquías y relaciones de especialización	Generalización	Herencia entre clases o actores	Cuando hay tipos y subtipos
Modelar flujo de datos	Modelo dirigido por datos	Entradas, procesos y salidas de datos	Sistemas centrados en la información más que en procesos

Ejemplo práctico

Supongamos que tienes un sistema web de biblioteca y quieres representarlo de diferentes formas:

- **Casos de uso:** Para ver que el *Usuario* puede *Buscar libro*, *Reservar libro* y *Devolver libro*, y que el *Bibliotecario* puede *Gestionar catálogo*.
- **Clases:** Para definir entidades como *Libro*, *Usuario*, *Préstamo*, sus atributos y relaciones.
- **Actividades:** Para mostrar el flujo desde que un usuario busca un libro hasta que confirma la reserva.
- **Secuencia:** Para detallar cómo interactúa el usuario con la interfaz, el backend y la base de datos.
- **Estados:** Para mostrar cómo un libro pasa de *Disponible* → *Reservado* → *Prestado* → *Devuelto*.
- **Contexto:** Para visualizar el sistema de biblioteca conectado con el *Ministerio de Educación*, *Plataforma de pagos*, etc.

Tabla de decisión para elegir diagramas UML

Objetivo / Pregunta que me hago sobre el sistema	Diagrama UML recomendado	Justificación
¿Quiénes usan el sistema y para qué?	Casos de uso	Identifica actores externos y funciones principales antes de cualquier diseño.
¿Qué partes componen el sistema y cómo se relacionan?	Diagrama de clases	Define la estructura interna (entidades, atributos, métodos, relaciones).
¿Cómo fluye el proceso paso a paso?	Diagrama de actividades	Permite visualizar el orden de tareas y decisiones que se toman.
¿En qué orden se comunican los componentes en un escenario específico?	Diagrama de secuencia	Muestra la interacción en tiempo real entre participantes y objetos.
¿Cómo cambia el estado de un elemento clave según eventos?	Diagrama de estados	Ideal para objetos con ciclo de vida definido (pedidos, usuarios, productos).
¿Qué sistemas externos están conectados y dónde termina mi sistema?	Diagrama de contexto	Define límites y conexiones con otros sistemas.
¿Existen jerarquías o especializaciones?	Jerarquía de generalización	Muestra herencia o subtipos (usuarios → estudiante, profesor).
¿Quiero centrarme en entradas, procesos y salidas de datos?	Modelo dirigido por datos	Útil en sistemas donde lo más importante es el flujo de información.

Prácticas

1. Sistema de Biblioteca

Escenario:

El sistema permite que un usuario registrado busque libros por título, autor o categoría, pueda reservar un libro disponible y consultar sus préstamos activos. Un bibliotecario gestiona el catálogo, aprueba reservas y registra devoluciones.
Necesidad: Ver quién interactúa con el sistema y qué funciones principales hay

---

2. Clínica Médica MHC-PMS

Escenario:

El médico crea un registro clínico para cada paciente. Este registro contiene diagnósticos, tratamientos y citas programadas. El recepcionista agenda las citas y notifica a los pacientes. El administrador gestiona usuarios y reportes.
Necesidad: Describir la estructura y relaciones entre las entidades.

---

3. Plataforma de Pedidos de Comida (similar a Uber Eats)

Escenario:

El cliente inicia sesión, selecciona un restaurante, añade platos a su carrito y confirma el pedido. El repartidor recibe el pedido, lo recoge en el restaurante y lo entrega. El restaurante actualiza el estado de preparación.
Necesidad: Visualizar el orden de interacción entre componentes.

---

4. Tienda Online

Escenario:

El cliente navega por productos, añade artículos al carrito y procede al pago. El sistema verifica inventario y procesa el pago con una pasarela. Si el pago es exitoso, se genera una orden y se notifica al cliente.
Necesidad: Explicar el flujo paso a paso de un proceso.

---

5. Sistema de Reservas de Vuelos

Escenario:

El viajero busca vuelos, selecciona uno y realiza la reserva. El sistema confirma disponibilidad, calcula el costo y solicita el pago. Un agente de soporte puede modificar o cancelar reservas bajo petición del viajero.
Necesidad: Mostrar cómo cambia un objeto según eventos.

---

6. Sistema Académico

Escenario:

Un estudiante se matricula en cursos disponibles, consulta calificaciones y descarga certificados. El profesor publica notas y crea actividades. El administrador del sistema gestiona usuarios y asignaturas.
Necesidad: Representar el sistema y su entorno.

7. Sistema de Control de Inventario

Descripción:

El encargado de bodega registra entradas y salidas de productos. El sistema actualiza el inventario y genera alertas cuando un producto está por agotarse. El gerente puede consultar reportes y aprobar pedidos de reposición.
Necesidad: Ver quién interactúa con el sistema y qué funciones principales hay.

---

8. Cajero Automático (ATM)

Descripción:

El cliente introduce su tarjeta y PIN. Puede consultar saldo, retirar dinero o depositar efectivo. El sistema verifica la cuenta en el banco y actualiza el saldo.
Necesidad: Explicar el flujo paso a paso de un proceso.

---

9. Sistema de Gestión Hotelera

Descripción:

El recepcionista registra huéspedes, asigna habitaciones y procesa pagos. El sistema permite al cliente hacer reservas en línea y al administrador generar reportes de ocupación.
Necesidad: Describir la estructura y relaciones entre las entidades.

---

10. Plataforma de Cursos Online

Descripción:

El estudiante se registra, selecciona un curso y accede a contenidos. El profesor sube materiales y califica tareas. El administrador gestiona usuarios y cursos.
Necesidad: Visualizar el orden de interacción entre componentes.

---

11. Sistema de Gestión de Consultas Legales

Descripción:

El cliente solicita una consulta. El abogado recibe la solicitud, la agenda y la atiende. El sistema registra el caso y permite al cliente revisar el historial de consultas.
Necesidad: Mostrar cómo cambia un objeto según eventos.

---

12. Aplicación de Mensajería

Descripción:

El usuario inicia sesión, envía mensajes de texto, imágenes o documentos. El servidor almacena y reenvía los mensajes al destinatario.
Necesidad: Representar el sistema y su entorno.

---

13. Sistema de Gestión de Taller Mecánico

Descripción:

El cliente agenda cita para reparar su vehículo. El mecánico realiza el diagnóstico, repara y actualiza el estado. El sistema emite la factura y registra el historial del vehículo.
Necesidad: Explicar el flujo paso a paso de un proceso.

---

14. Plataforma de Venta de Boletos para Eventos

Descripción:

El usuario busca eventos, selecciona asiento y compra boleto. El sistema verifica disponibilidad, procesa el pago y genera el boleto digital.
Necesidad: Visualizar el orden de interacción entre componentes.

---

15. Sistema de Gestión de Restaurante

Descripción:

El mesero toma pedidos y los envía a la cocina. El chef actualiza el estado del pedido. El cajero genera la factura y procesa el pago.
Necesidad: Describir la estructura y relaciones entre las entidades.

---

16. Sistema de Monitoreo de Energía

Descripción:

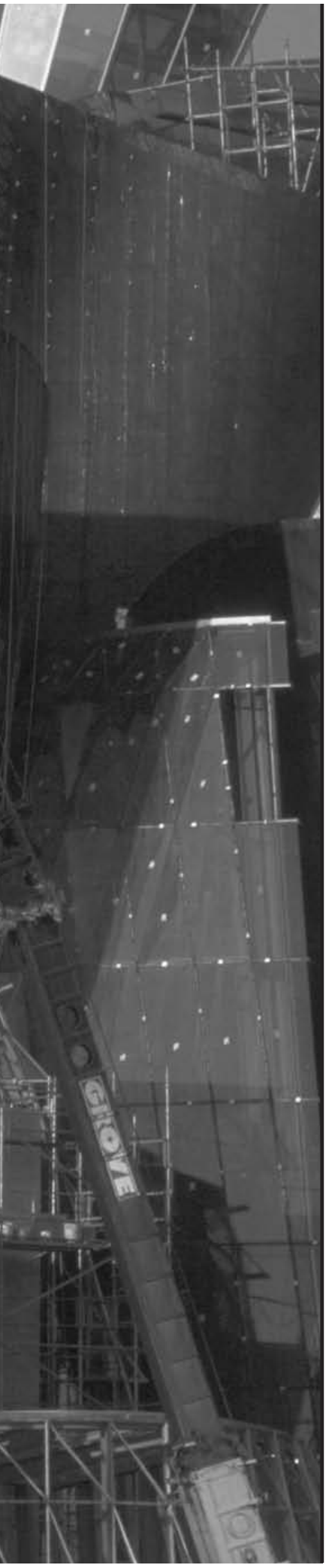
Los sensores envían datos de consumo eléctrico a un servidor. El sistema analiza el consumo y genera alertas si se superan los límites establecidos.
Necesidad: Representar el sistema y su entorno.

Ejemplo de prompt genérico

Genera el código PlantUML de un diagrama [tipo de diagrama UML] para el siguiente escenario:

[Descripción del escenario]

El diagrama debe reflejar todos los elementos mencionados y usar sintaxis válida de PlantUML.



6

Diseño arquitectónico

Objetivos

El objetivo de este capítulo es introducir los conceptos de arquitectura de software y diseño arquitectónico. Al estudiar este capítulo:

- comprenderá por qué es importante el diseño arquitectónico del software;
- conocerá las decisiones que deben tomarse sobre la arquitectura de software durante el proceso de diseño arquitectónico;
- asimilará la idea de los patrones arquitectónicos, formas bien reconocidas de organización de las arquitecturas del sistema, que pueden reutilizarse en los diseños del sistema;
- identificará los patrones arquitectónicos usados frecuentemente en diferentes tipos de sistemas de aplicación, incluidos los sistemas de procesamiento de transacción y los sistemas de procesamiento de lenguaje.

Contenido

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.

Para ayudar a comprender lo que se entiende por arquitectura del sistema, tome en cuenta la figura 6.1. En ella se presenta un modelo abstracto de la arquitectura para un sistema de robot de empaquetado, que indica los componentes que tienen que desarrollarse. Este sistema robótico empaca diferentes clases de objetos. Usa un componente de visión para recoger los objetos de una banda transportadora, identifica la clase de objeto y selecciona el tipo correcto de empaque. Luego, el sistema mueve los objetos que va a empacar de la banda transportadora de entrega y coloca los objetos empacados en otro transportador. El modelo arquitectónico presenta dichos componentes y los vínculos entre ellos.

En la práctica, hay un significativo traslape entre los procesos de ingeniería de requerimientos y el diseño arquitectónico. De manera ideal, una especificación de sistema no debe incluir cierta información de diseño. Esto no es realista, excepto para sistemas muy pequeños. La descomposición arquitectónica es por lo general necesaria para estructurar y organizar la especificación. Por lo tanto, como parte del proceso de ingeniería de requerimientos, usted podría proponer una arquitectura de sistema abstracta donde se asocien grupos de funciones de sistemas o características con componentes o subsistemas a gran escala. Luego, puede usar esta descomposición para discutir con los participantes sobre los requerimientos y las características del sistema.

Las arquitecturas de software se diseñan en dos niveles de abstracción, que en este texto se llaman *arquitectura en pequeño* y *arquitectura en grande*:

1. La arquitectura en pequeño se interesa por la arquitectura de programas individuales. En este nivel, uno se preocupa por la forma en que el programa individual se separa en componentes. Este capítulo se centra principalmente en arquitecturas de programa.

2. La arquitectura en grande se interesa por la arquitectura de sistemas empresariales complejos que incluyen otros sistemas, programas y componentes de programa. Tales sistemas empresariales se distribuyen a través de diferentes computadoras, que diferentes compañías administran y poseen. En los capítulos 18 y 19 se cubren las arquitecturas grandes; en ellos se estudiarán las arquitecturas de los sistemas distribuidos.

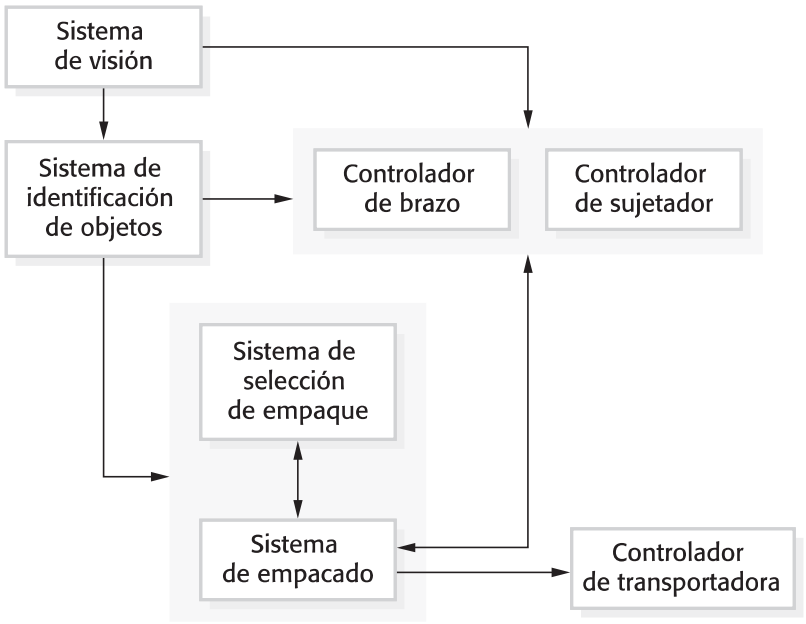


Figura 6.1 Arquitectura de un sistema de control para un robot empacador

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. Como se explica en el capítulo 16, es posible desarrollar arquitecturas de línea de productos donde la misma arquitectura se reutilice mediante una amplia gama de sistemas relacionados.

Hofmeister y sus colaboradores (2000) proponen que una arquitectura de software sirve en primer lugar como un plan de diseño para la negociación de requerimientos de sistema y, en segundo lugar, como un medio para establecer discusiones con clientes, desarrolladores y administradores. También sugieren que es una herramienta esencial para la administración de la complejidad. Oculta los detalles y permite a los diseñadores enfocarse en las abstracciones clave del sistema.

Las arquitecturas de sistemas se modelan con frecuencia usando diagramas de bloques simples, como en la figura 6.1. Cada recuadro en el diagrama representa un componente. Los recuadros dentro de recuadros indican que el componente se dividió en subcomponentes. Las flechas significan que los datos y/o señales de control pasan de un componente a otro en la dirección de las flechas. Hay varios ejemplos de este tipo de modelo arquitectónico en el catálogo de arquitectura de software de Booch (Booch, 2009).

Los diagramas de bloque presentan una imagen de alto nivel de la estructura del sistema e incluyen fácilmente a individuos de diferentes disciplinas que intervienen en el proceso de desarrollo del sistema. No obstante su amplio uso, Bass y sus colaboradores (2003) no están de acuerdo con los diagramas de bloque informales para describir una arquitectura. Afirman que tales diagramas informales son representaciones arquitectónicas deficientes, pues no muestran ni el tipo de relaciones entre los componentes del sistema ni las propiedades externamente visibles de los componentes.

Las aparentes contradicciones entre práctica y teoría arquitectónica surgen porque hay dos formas en que se utiliza un modelo arquitectónico de un programa:

1. *Como una forma de facilitar la discusión acerca del diseño del sistema* Una visión arquitectónica de alto nivel de un sistema es útil para la comunicación con los participantes de un sistema y la planeación del proyecto, ya que no se satura con detalles. Los participantes pueden relacionarse con él y entender una visión abstracta del sistema. En tal caso, analizan el sistema como un todo sin confundirse por los detalles. El modelo arquitectónico identifica los componentes clave que se desarrollarán, de modo que los administradores pueden asignar a individuos para planear el desarrollo de dichos sistemas.

2. *Como una forma de documentar una arquitectura que se haya diseñado* La meta aquí es producir un modelo de sistema completo que muestre los diferentes componentes en un sistema, sus interfaces y conexiones. El argumento para esto es que tal descripción arquitectónica detallada facilita la comprensión y la evolución del sistema.

Los diagramas de bloque son una forma adecuada para describir la arquitectura del sistema durante el proceso de diseño, pues son una buena manera de soportar las comunicaciones entre las personas involucradas en el proceso. En muchos proyectos, suele ser la única documentación arquitectónica que existe. Sin embargo, si la arquitectura de un sistema debe documentarse ampliamente, entonces es mejor usar una notación con semántica bien definida para la descripción arquitectónica. No obstante, tal como se estudia en la sección 6.2, algunas personas consideran que la documentación detallada ni es útil ni vale realmente la pena el costo de su desarrollo.

6.1 Decisiones en el diseño arquitectónico

El diseño arquitectónico es un proceso creativo en el cual se diseña una organización del sistema que cubrirá los requerimientos funcionales y no funcionales de éste. Puesto que se trata de un proceso creativo, las actividades dentro del proceso dependen del tipo de sistema que se va a desarrollar, los antecedentes y la experiencia del arquitecto del sistema, así como de los requerimientos específicos del sistema. Por lo tanto, es útil pensar en el diseño arquitectónico como un conjunto de decisiones a tomar en vez de una secuencia de actividades.

Durante el proceso de diseño arquitectónico, los arquitectos del sistema deben tomar algunas decisiones estructurales que afectarán profundamente el sistema y su proceso de desarrollo. Con base en su conocimiento y experiencia, deben considerar las siguientes preguntas fundamentales sobre el sistema:

1. ¿Existe alguna arquitectura de aplicación genérica que actúe como plantilla para el sistema que se está diseñando?
2. ¿Cómo se distribuirá el sistema a través de algunos núcleos o procesadores?
3. ¿Qué patrones o estilos arquitectónicos pueden usarse?
4. ¿Cuál será el enfoque fundamental usado para estructurar el sistema?
5. ¿Cómo los componentes estructurales en el sistema se separarán en subcomponentes?
6. ¿Qué estrategia se usará para controlar la operación de los componentes en el sistema?
7. ¿Cuál organización arquitectónica es mejor para entregar los requerimientos no funcionales del sistema?
8. ¿Cómo se evaluará el diseño arquitectónico?
9. ¿Cómo se documentará la arquitectura del sistema?

Aunque cada sistema de software es único, los sistemas en el mismo dominio de aplicación tienen normalmente arquitecturas similares que reflejan los conceptos fundamentales del dominio. Por ejemplo, las líneas de producto de aplicación son aplicaciones que se construyen en torno a una arquitectura central con variantes que cubren requerimientos específicos del cliente. Cuando se diseña una arquitectura de sistema, debe decidirse qué tienen en común el sistema y las clases de aplicación más amplias, con la finalidad de determinar cuánto conocimiento se puede reutilizar de dichas arquitecturas de aplicación. En la sección 6.4 se estudian las arquitecturas de aplicación genéricas y en el capítulo 16 las líneas de producto de aplicación.

Para sistemas embebidos y sistemas diseñados para computadoras personales, por lo general, hay un solo procesador y no tendrá que diseñar una arquitectura distribuida para el sistema. Sin embargo, los sistemas más grandes ahora son sistemas distribuidos donde el software de sistema se distribuye a través de muchas y diferentes computadoras. La elección de arquitectura de distribución es una decisión clave que afecta el rendimiento y

la fiabilidad del sistema. Éste es un tema importante por derecho propio y se trata por separado en el capítulo 18.

La arquitectura de un sistema de software puede basarse en un patrón o un estilo arquitectónico particular. Un patrón arquitectónico es una descripción de una organización del sistema (Garlan y Shaw, 1993), tal como una organización cliente-servidor o una arquitectura por capas. Los patrones arquitectónicos captan la esencia de una arquitectura que se usó en diferentes sistemas de software. Usted tiene que conocer tanto los patrones comunes, en que éstos se usen, como sus fortalezas y debilidades cuando se tomen decisiones sobre la arquitectura de un sistema. En la sección 6.3 se analizan algunos patrones de uso frecuente.

La noción de un estilo arquitectónico de Garlan y Shaw (estilo y patrón llegaron a significar lo mismo) cubre las preguntas 4 a 6 de la lista anterior. Es necesario elegir la estructura más adecuada, como cliente-servidor o estructura en capas, que le permita satisfacer los requerimientos del sistema. Para descomponer las unidades del sistema estructural, usted opta por la estrategia de separar los componentes en subcomponentes. Los enfoques que pueden usarse permiten la implementación de diferentes tipos de arquitectura. Finalmente, en el proceso de modelado de control, se toman decisiones sobre cómo se controla la ejecución de componentes. Se desarrolla un modelo general de las relaciones de control entre las diferentes partes del sistema.

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 éste 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.

Evidentemente, hay un conflicto potencial entre algunas de estas arquitecturas. Por ejemplo, usar componentes grandes mejora el rendimiento, y utilizar componentes pequeños de grano fino aumenta la mantenibilidad. Si tanto el rendimiento como la mantenibilidad son requerimientos importantes del sistema, entonces debe encontrarse algún compromiso. Esto en ocasiones se logra usando diferentes patrones o estilos arquitectónicos para distintas partes del sistema.

Evaluar un diseño arquitectónico es difícil porque la verdadera prueba de una arquitectura es qué tan bien el sistema cubre sus requerimientos funcionales y no funcionales cuando está en uso. Sin embargo, es posible hacer cierta evaluación al comparar el diseño contra arquitecturas de referencia o patrones arquitectónicos genéricos. Para ayudar con la evaluación arquitectónica, también puede usarse la descripción de Bosch (2000) de las características no funcionales de los patrones arquitectónicos.

6.2 Vistas arquitectónicas

En la introducción a este capítulo se explicó que los modelos arquitectónicos de un sistema de software sirven para enfocar la discusión sobre los requerimientos o el diseño del software. De manera alternativa, pueden emplearse para documentar un diseño, de modo que se usen como base en el diseño y la implementación más detallados, así como en la evolución futura del sistema. En esta sección se estudian dos temas que son relevantes para ambos:

1. ¿Qué vistas o perspectivas son útiles al diseñar y documentar una arquitectura del sistema?
2. ¿Qué notaciones deben usarse para describir modelos arquitectónicos?

Es imposible representar toda la información relevante sobre la arquitectura de un sistema en un solo modelo arquitectónico, ya que cada uno presenta únicamente una vista o perspectiva del sistema. Ésta puede mostrar cómo un sistema se descompone en módulos, cómo interactúan los procesos de tiempo de operación o las diferentes formas en que los componentes del sistema se distribuyen a través de una red. Todo ello es útil en diferentes momentos de manera que, para el diseño y la documentación, por lo general se necesita presentar múltiples vistas de la arquitectura de software.

Existen diferentes opiniones relativas a qué vistas se requieren. Krutchen (1995), en su bien conocido modelo de vista 4+1 de la arquitectura de software, sugiere que deben existir cuatro vistas arquitectónicas fundamentales, que se relacionan usando casos de uso o escenarios. Las vistas que él sugiere son:

1. Una vista lógica, que indique las abstracciones clave en el sistema como objetos o clases de objeto. En este tipo de vista se tienen que relacionar los requerimientos del sistema con entidades.

2. Una vista de proceso, que muestre cómo, en el tiempo de operación, el sistema está compuesto de procesos en interacción. Esta vista es útil para hacer juicios acerca de las características no funcionales del sistema, como el rendimiento y la disponibilidad.
3. Una vista de desarrollo, que muestre cómo el software está descompuesto para su desarrollo, esto es, indica la descomposición del software en elementos que se implementen mediante un solo desarrollador o equipo de desarrollo. Esta vista es útil para administradores y programadores de software.
4. Una vista física, que exponga el hardware del sistema y cómo los componentes de software se distribuyen a través de los procesadores en el sistema. Esta vista es útil para los ingenieros de sistemas que planean una implementación de sistema.

Hofmeister y sus colaboradores (2000) sugieren el uso de vistas similares, pero a éstas agregan la noción de vista conceptual. Esta última es una vista abstracta del sistema que puede ser la base para descomponer los requerimientos de alto nivel en especificaciones más detalladas, ayudar a los ingenieros a tomar decisiones sobre componentes que puedan reutilizarse, y representar una línea de producto (que se estudia en el capítulo 16) en vez de un solo sistema. La figura 6.1, que describe la arquitectura de un robot de empacado, es un ejemplo de una vista conceptual del sistema.

En la práctica, las vistas conceptuales casi siempre se desarrollan durante el proceso de diseño y se usan para apoyar la toma de decisiones arquitectónicas. Son una forma de comunicar a diferentes participantes la esencia de un sistema. Durante el proceso de diseño, también pueden desarrollarse algunas de las otras vistas, al tiempo que se discuten diferentes aspectos del sistema, aunque no haya necesidad de una descripción completa desde todas las perspectivas. Además se podrían asociar patrones arquitectónicos, estudiados en la siguiente sección, con las diferentes vistas de un sistema.

Hay diferentes opiniones respecto de si los arquitectos de software deben o no usar el UML para una descripción arquitectónica (Clements *et al*., 2002). Un estudio en 2006 (Lange *et al*., 2006) demostró que, cuando se usa el UML, se aplica principalmente en una forma holgada e informal. Los autores de dicho ensayo argumentan que esto era incorrecto. El autor no está de acuerdo con esta visión. El UML se diseñó para describir sistemas orientados a objetos y, en la etapa de diseño arquitectónico, uno quiere describir con frecuencia sistemas en un nivel superior de abstracción. Las clases de objetos están muy cerca de la implementación, como para ser útiles en la descripción arquitectónica.

Para el autor, el UML no es útil durante el proceso de diseño en sí y prefiere notaciones informales que sean más rápidas de escribir y puedan dibujarse fácilmente en un pizarrón. El UML es de más valor cuando se documenta una arquitectura a detalle o se usa un desarrollo dirigido por modelo, como se estudió en el capítulo 5.

Algunos investigadores proponen el uso de lenguajes de descripción arquitectónica (ADL, por las siglas de *Architectural Description Languages*) más especializados (Bass *et al*., 2003) para describir arquitecturas del sistema. Los elementos básicos de los ADL son componentes y conectores, e incluyen reglas y lineamientos para arquitecturas bien formadas. Sin embargo, debido a su naturaleza especializada, los expertos de dominio y aplicación tienen dificultad para entender y usar los ADL. Esto dificulta la valoración de su utilidad para la ingeniería práctica del software. Los ADL diseñados para un dominio particular (por ejemplo, sistemas automotores) pueden usarse como una base

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.

Figura 6.2 Patrón modelo de vista del controlador (MVC)

para el desarrollo dirigido por modelo. Sin embargo, se considera que los modelos y las notaciones informales, como el UML, seguirán siendo las formas de uso más común para documentar las arquitecturas del sistema.

Los usuarios de métodos ágiles afirman que, por lo general, no se utiliza la documentación detallada del diseño. Por lo tanto, desarrollarla es un desperdicio de tiempo y dinero. El autor está en gran medida de acuerdo con esta visión y considera que, para la mayoría de los sistemas, no vale la pena desarrollar una descripción arquitectónica detallada desde estas cuatro perspectivas. Uno debe desarrollar las vistas que sean útiles para la comunicación sin preocuparse si la documentación arquitectónica está completa o no. Sin embargo, una excepción es al desarrollar sistemas críticos, cuando es necesario realizar un análisis de confiabilidad detallado del sistema. Tal vez se deba convencer a reguladores externos de que el sistema se hizo conforme a sus regulaciones y, en consecuencia, puede requerirse una documentación arquitectónica completa.

6.3 Patrones arquitectónicos

La idea de los patrones como una forma de presentar, compartir y reutilizar el conocimiento sobre los sistemas de software se usa ahora ampliamente. El origen de esto fue la publicación de un libro acerca de patrones de diseño orientados a objetos (Gamma *et al.*, 1995), que incitó el desarrollo de otros tipos de patrón, como los patrones para el diseño organizacional (Coplien y Harrison, 2004), patrones de usabilidad (Usability Group, 1998), interacción (Martin y Sommerville, 2004), administración de la configuración

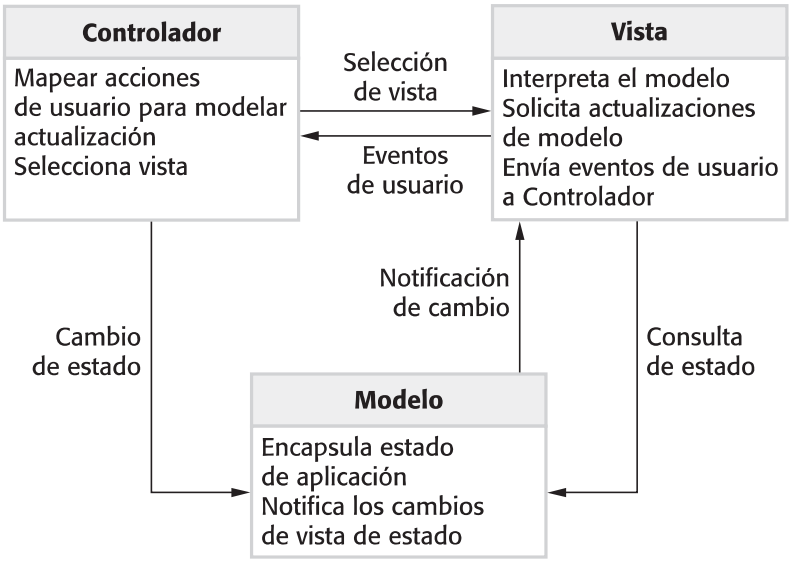


Figura 6.3 La organización del MVC

(Berczuk y Appleton, 2002), etcétera. Los patrones arquitectónicos se propusieron en la década de 1990, con el nombre de "estilos arquitectónicos" (Shaw y Garlan, 1996), en una serie de cinco volúmenes de manuales sobre arquitectura de software orientada a patrones, publicados entre 1996 y 2007 (Buschmann *et al.*, 1996; Buschmann *et al.*, 2007a; Buschmann *et al.*, 2007b; Kircher y Jain, 2004; Schmidt *et al.*, 2000).

En esta sección se introducen los patrones arquitectónicos y se describe brevemente una selección de patrones arquitectónicos de uso común en diferentes tipos de sistemas. Para más información de patrones y su uso, debe remitirse a los manuales de patrones publicados.

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, la figura 6.2 describe 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). También debe incluir información sobre cuándo hay que usar el patrón, así como sobre sus ventajas y desventajas. En las figuras 6.3 y 6.4 se presentan los modelos gráficos de la arquitectura asociada con el patrón MVC. En ellas se muestra la arquitectura desde diferentes vistas: la figura 6.3 es una vista conceptual; en tanto que la figura 6.4 ilustra una posible arquitectura en tiempo de operación, cuando este patrón se usa para el manejo de la interacción en un sistema basado en la Web.

En una breve sección de un capítulo general es imposible describir todos los patrones genéricos que se usan en el desarrollo de software. En cambio, se presentan algunos ejemplos seleccionados de patrones que se utilizan ampliamente y captan buenos principios de diseño arquitectónico. En las páginas Web del libro se incluyen más ejemplos acerca de patrones arquitectónicos genéricos.

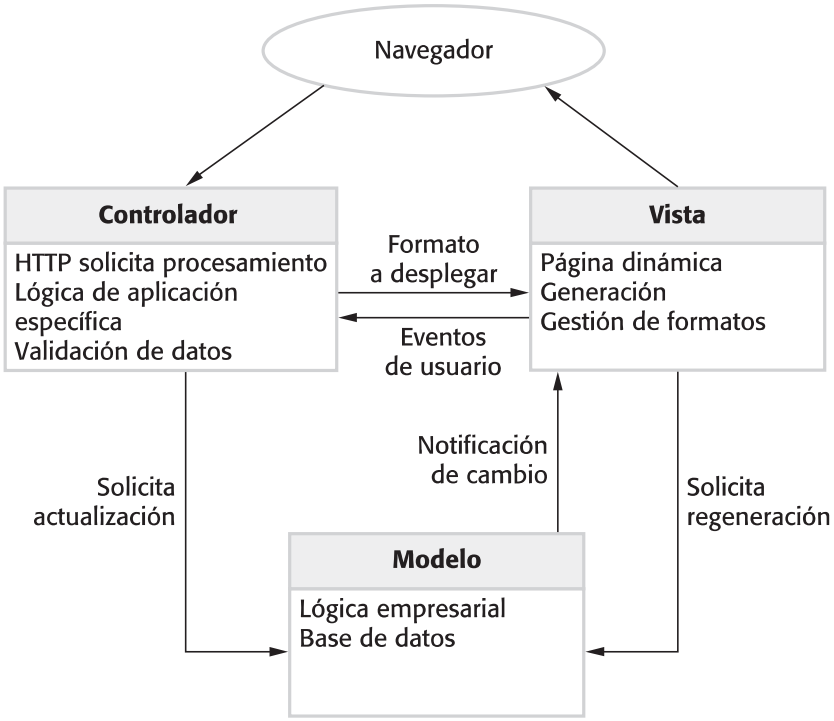


Figura 6.4 Arquitectura de aplicación Web con el patrón MVC

6.3.1 Arquitectura en capas

Las nociones de separación e independencia son fundamentales para el diseño arquitectónico porque permiten localizar cambios. El patrón MVC, que se muestra en la figura 6.2, separa elementos de un sistema, permitiéndoles cambiar de forma independiente. Por ejemplo, agregar una nueva vista o cambiar una vista existente puede hacerse sin modificación alguna a los datos subyacentes en el modelo. El patrón de arquitectura en capas es otra forma de lograr separación e independencia. Este patrón se ilustra en la figura 6.5. Aquí, la funcionalidad del sistema está organizada en capas separadas, y cada una se apoya sólo en las facilidades y los servicios ofrecidos por la capa inmediatamente debajo de ella.

Este enfoque en capas soporta el desarrollo incremental de sistemas. Conforme se desarrolla una capa, algunos de los servicios proporcionados por esta capa deben quedar a disposición de los usuarios. La arquitectura también es cambiable y portátil. En tanto su interfaz no varíe, una capa puede sustituirse por otra equivalente. Más aún, cuando las interfaces de capa cambian o se agregan nuevas facilidades a una capa, sólo resulta afectada la capa adyacente. A medida que los sistemas en capas localizan dependencias de máquina en capas más internas, se facilita el ofrecimiento de implementaciones multiplataforma de un sistema de aplicación. Sólo las capas más internas dependientes de la máquina deben reimplantarse para considerar las facilidades de un sistema operativo o base de datos diferentes.

La figura 6.6 es un ejemplo de una arquitectura en capas con cuatro capas. La capa inferior incluye software de soporte al sistema, por lo general soporte de base de datos y sistema operativo. La siguiente capa es la de aplicación, que comprende los componentes relacionados con la funcionalidad de la aplicación, así como los componentes de utilidad que usan otros componentes de aplicación. La tercera capa se relaciona con la gestión de interfaz del usuario y con brindar autenticación y autorización al usuario, mientras que la

Nombre	Arquitectura en capas
Descripción	Organiza el sistema en capas con funcionalidad relacionada con cada capa. Una capa da servicios a la capa de encima, de modo que las capas de nivel inferior representan servicios núcleo que es probable se utilicen a lo largo de todo el sistema. Véase la figura 6.6.
Ejemplo	Un modelo en capas de un sistema para compartir documentos con derechos de autor se tiene en diferentes bibliotecas, como se ilustra en la figura 6.7.
Cuándo se usa	Se usa al construirse nuevas facilidades encima de los sistemas existentes; cuando el desarrollo se dispersa a través de varios equipos de trabajo, y cada uno es responsable de una capa de funcionalidad; cuando exista un requerimiento para seguridad multinivel.
Ventajas	Permite la sustitución de capas completas en tanto se conserve la interfaz. Para aumentar la confiabilidad del sistema, en cada capa pueden incluirse facilidades redundantes (por ejemplo, autenticación).
Desventajas	En la práctica, suele ser difícil ofrecer una separación limpia entre capas, y es posible que una capa de nivel superior deba interactuar directamente con capas de nivel inferior, en vez de que sea a través de la capa inmediatamente abajo de ella. El rendimiento suele ser un problema, debido a múltiples niveles de interpretación de una solicitud de servicio mientras se procesa en cada capa.

Figura 6.5 Patrón de arquitectura en capas

capa superior proporciona facilidades de interfaz de usuario. Desde luego, es arbitrario el número de capas. Cualquiera de las capas en la figura 6.6 podría dividirse en dos o más capas.

La figura 6.7 es un ejemplo de cómo puede aplicarse este patrón de arquitectura en capas a un sistema de biblioteca llamado LIBSYS, que permite el acceso electrónico controlado a material con derechos de autor de un conjunto de bibliotecas universitarias. Tiene una arquitectura de cinco capas y, en la capa inferior, están las bases de datos individuales en cada biblioteca.

En la figura 6.17 (que se encuentra en la sección 6.4) se observa otro ejemplo de patrón de arquitectura en capas. Muestra la organización del sistema para atención a la salud mental (MHC-PMS) que se estudió en capítulos anteriores.

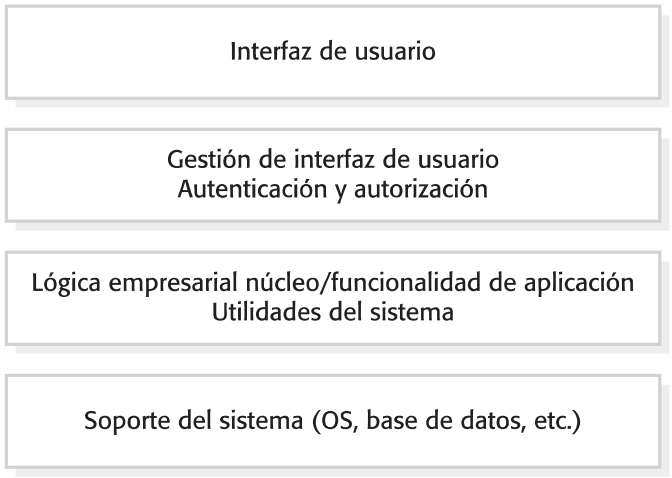


Figura 6.6 Arquitectura genérica en capas

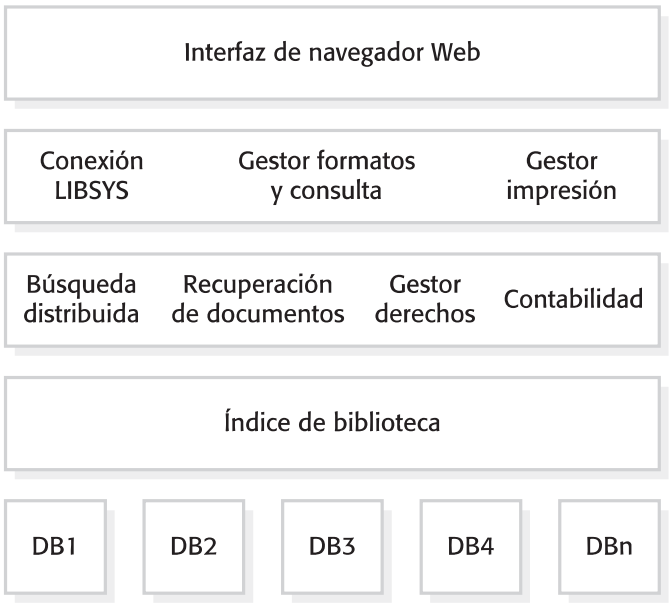


Figura 6.7 Arquitectura del sistema LIBSYS

6.3.2 Arquitectura de repositorio

Los patrones de arquitectura en capas y MVC son ejemplos de patrones en que la vista presentada es la organización conceptual de un sistema. El siguiente ejemplo, el patrón de repositorio (figura 6.8), describe cómo comparte datos un conjunto de componentes en interacción.

La mayoría de los sistemas que usan grandes cantidades de datos se organizan sobre una base de datos o un repositorio compartido. Por lo tanto, este modelo es adecuado

Figura 6.8 El patrón de repositorio

Nombre	Repositorio
Descripción	Todos los datos en un sistema se gestionan en un repositorio central, accesible a todos los componentes del sistema. Los componentes no interactúan directamente, sino tan sólo a través del repositorio.
Ejemplo	La figura 6.9 es un ejemplo de un IDE donde los componentes usan un repositorio de información de diseño de sistema. Cada herramienta de software genera información que, en ese momento, está disponible para uso de otras herramientas.
Cuándo se usa	Este patrón se usa cuando se tiene un sistema donde los grandes volúmenes de información generados deban almacenarse durante mucho tiempo. También puede usarse en sistemas dirigidos por datos, en los que la inclusión de datos en el repositorio active una acción o herramienta.
Ventajas	Los componentes pueden ser independientes, no necesitan conocer la existencia de otros componentes. Los cambios hechos por un componente se pueden propagar hacia todos los componentes. La totalidad de datos se puede gestionar de manera consistente (por ejemplo, respaldos realizados al mismo tiempo), pues todos están en un lugar.
Desventajas	El repositorio es un punto de falla único, de modo que los problemas en el repositorio afectan a todo el sistema. Es posible que haya ineficiencias al organizar toda la comunicación a través del repositorio. Quizá sea difícil distribuir el repositorio por medio de varias computadoras.

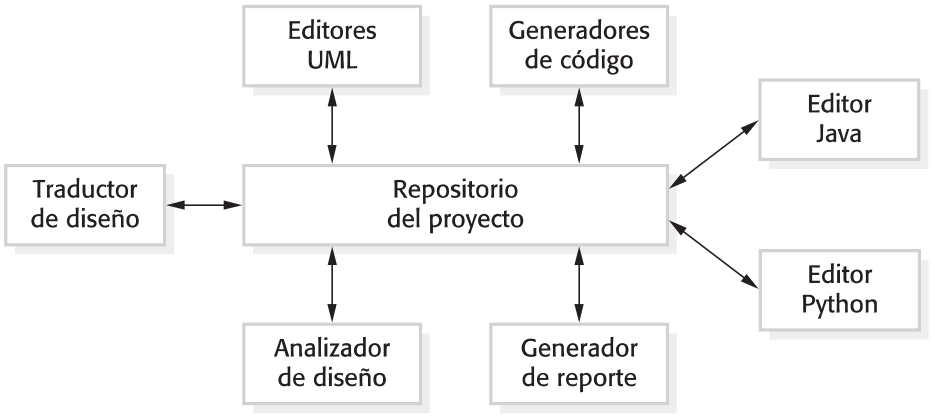


Figura 6.9 Arquitectura de repositorio para un IDE

para aplicaciones en las que un componente genere datos y otro los use. Los ejemplos de este tipo de sistema incluyen sistemas de comando y control, sistemas de información administrativa, sistemas CAD y entornos de desarrollo interactivo para software.

La figura 6.9 ilustra una situación en la que puede usarse un repositorio. Este diagrama muestra un IDE que incluye diferentes herramientas para soportar desarrollo dirigido por modelo. En este caso, el repositorio puede ser un entorno controlado por versión (como se estudia en el capítulo 25) que hace un seguimiento de los cambios al software y permite regresar (*rollback*) a versiones anteriores.

Organizar herramientas alrededor de un repositorio es una forma eficiente de compartir grandes cantidades de datos. No hay necesidad de transmitir explícitamente datos de un componente a otro. Sin embargo, los componentes deben operar en torno a un modelo de repositorio de datos acordado. Inevitablemente, éste es un compromiso entre las necesidades específicas de cada herramienta y sería difícil o imposible integrar nuevos componentes, si sus modelos de datos no se ajustan al esquema acordado. En la práctica, llega a ser complicado distribuir el repositorio sobre un número de máquinas. Aunque es posible distribuir un repositorio lógicamente centralizado, puede haber problemas con la redundancia e inconsistencia de los datos.

En el ejemplo que se muestra en la figura 6.9, el repositorio es pasivo, y el control es responsabilidad de los componentes que usan el repositorio. Un enfoque alternativo, que se derivó para sistemas IA, utiliza un modelo "blackboard" (pizarrón) que activa componentes cuando los datos particulares se tornan disponibles. Esto es adecuado cuando la forma de los datos del repositorio está menos estructurada. Las decisiones sobre cuál herramienta activar puede hacerse sólo cuando se hayan analizado los datos. Este modelo lo introdujo Nii (1986). Bosch (2000) incluye un buen análisis de cómo este estilo se relaciona con los atributos de calidad del sistema.

6.3.3 Arquitectura cliente-servidor

El patrón de repositorio se interesa por la estructura estática de un sistema sin mostrar su organización en tiempo de operación. El siguiente ejemplo ilustra una organización en tiempo de operación, de uso muy común para sistemas distribuidos. En la figura 6.10 se describe el patrón cliente-servidor.

Nombre	Cliente-servidor
Descripción	En una arquitectura cliente-servidor, la funcionalidad del sistema se organiza en servicios, y cada servicio lo entrega un servidor independiente. Los clientes son usuarios de dichos servicios y para utilizarlos ingresan a los servidores.
Ejemplo	La figura 6.11 es un ejemplo de una filmoteca y videoteca (videos/DVD) organizada como un sistema cliente-servidor.
Cuándo se usa	Se usa cuando, desde varias ubicaciones, se tiene que ingresar a los datos en una base de datos compartida. Como los servidores se pueden replicar, también se usan cuando la carga de un sistema es variable.
Ventajas	La principal ventaja de este modelo es que los servidores se pueden distribuir a través de una red. La funcionalidad general (por ejemplo, un servicio de impresión) estaría disponible a todos los clientes, así que no necesita implementarse en todos los servicios.
Desventajas	Cada servicio es un solo punto de falla, de modo que es susceptible a ataques de rechazo de servicio o a fallas del servidor. El rendimiento resultará impredecible porque depende de la red, así como del sistema. Quizás haya problemas administrativos cuando los servidores sean propiedad de diferentes organizaciones.

Figura 6.10 Patrón cliente-servidor

Un sistema que sigue el patrón cliente-servidor se organiza como un conjunto de servicios y servidores asociados, y de clientes que acceden y usan los servicios. Los principales componentes de este modelo son:

1. Un conjunto de servidores que ofrecen servicios a otros componentes. Ejemplos de éstos incluyen servidores de impresión; servidores de archivo que brindan servicios de administración de archivos, y un servidor compilador, que proporciona servicios de compilación de lenguaje de programación.
2. Un conjunto de clientes que solicitan los servicios que ofrecen los servidores. Habrá usualmente varias instancias de un programa cliente que se ejecuten de manera concurrente en diferentes computadoras.
3. Una red que permite a los clientes acceder a dichos servicios. La mayoría de los sistemas cliente-servidor se implementan como sistemas distribuidos, conectados mediante protocolos de Internet.

Las arquitecturas cliente-servidor se consideran a menudo como arquitecturas de sistemas distribuidos; sin embargo, el modelo lógico de servicios independientes que opera en servidores separados puede implementarse en una sola computadora. De nuevo, un beneficio importante es la separación e independencia. Los servicios y servidores pueden cambiar sin afectar otras partes del sistema.

Es posible que los clientes deban conocer los nombres de los servidores disponibles, así como los servicios que proporcionan. Sin embargo, los servidores no necesitan conocer la identidad de los clientes o cuántos clientes acceden a sus servicios. Los clientes acceden a los servicios que proporciona un servidor, a través de llamadas a procedimiento remoto usando un protocolo solicitud-respuesta, como el protocolo http utilizado

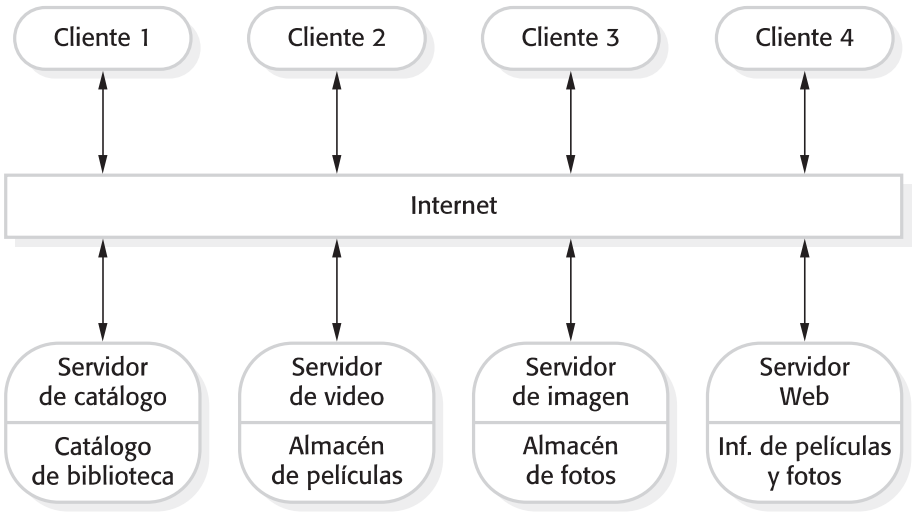


Figura 6.11 Arquitectura cliente-servidor para una filmoteca

en la WWW. En esencia, un cliente realiza una petición a un servidor y espera hasta que recibe una respuesta.

La figura 6.11 es un ejemplo de un sistema que se basa en el modelo cliente-servidor. Se trata de un sistema multiusuario basado en la Web, para ofrecer un repertorio de películas y fotografías. En este sistema, varios servidores manejan y despliegan los diferentes tipos de medios. Los cuadros de video necesitan transmitirse rápidamente y en sincronía, aunque a una resolución relativamente baja. Tal vez estén comprimidos en un almacén, de manera que el servidor de video puede manipular en diferentes formatos la compresión y descompresión del video. Sin embargo, las imágenes fijas deben conservarse en una resolución alta, por lo que es adecuado mantenerlas en un servidor independiente.

El catálogo debe manejar una variedad de consultas y ofrecer vínculos hacia el sistema de información Web, que incluye datos acerca de las películas y los videos, así

Figura 6.12 Patrón de tubería y filtro (*pipe and filter*)

Nombre	Tubería y filtro (*pipe and filter*)
Descripción	El procesamiento de datos en un sistema se organiza de forma que cada componente de procesamiento (filtro) sea discreto y realice un tipo de transformación de datos. Los datos fluyen (como en una tubería) de un componente a otro para su procesamiento.
Ejemplo	La figura 6.13 es un ejemplo de un sistema de tubería y filtro usado para el procesamiento de facturas.
Cuándo se usa	Se suele utilizar en aplicaciones de procesamiento de datos (tanto basadas en lotes [*batch*] como en transacciones), donde las entradas se procesan en etapas separadas para generar salidas relacionadas.
Ventajas	Fácil de entender y soporta reutilización de transformación. El estilo del flujo de trabajo coincide con la estructura de muchos procesos empresariales. La evolución al agregar transformaciones es directa. Puede implementarse como un sistema secuencial o como uno concurrente.
Desventajas	El formato para la transferencia de datos debe acordarse entre las transformaciones que se comunican. Cada transformación debe analizar sus entradas y sintetizar sus salidas al formato acordado. Esto aumenta la carga del sistema, y puede significar que sea imposible reutilizar transformaciones funcionales que usen estructuras de datos incompatibles.

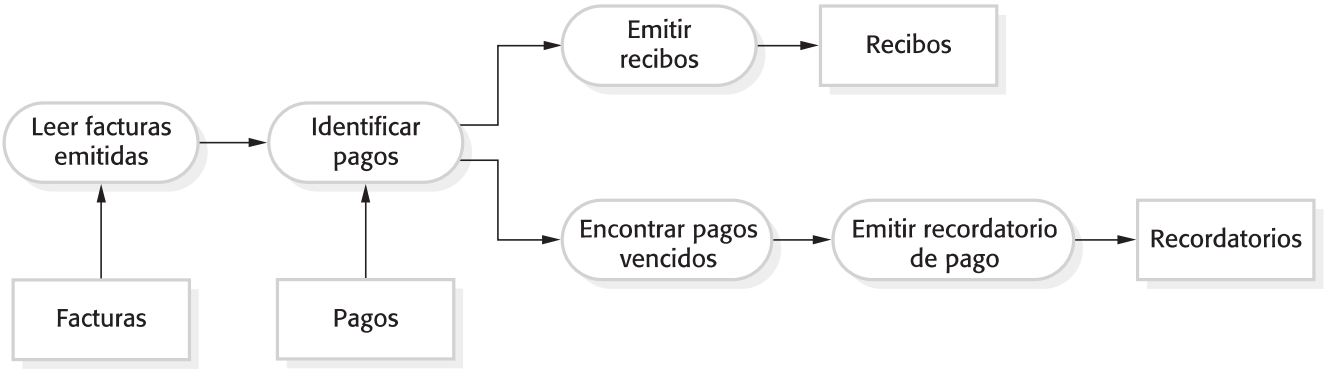


Figura 6.13 Ejemplo de arquitectura de tubería y filtro

como un sistema de comercio electrónico que soporte la venta de fotografías, películas y videos. El programa cliente es simplemente una interfaz integrada de usuario, construida mediante un navegador Web, para acceder a dichos servicios.

La ventaja más importante del modelo cliente-servidor consiste en que es una arquitectura distribuida. Éste puede usarse de manera efectiva en sistemas en red con distintos procesadores distribuidos. Es fácil agregar un nuevo servidor e integrarlo al resto del sistema, o bien, actualizar de manera clara servidores sin afectar otras partes del sistema. En el capítulo 18 se estudian las arquitecturas distribuidas, incluidas las arquitecturas cliente-servidor y las arquitecturas de objeto distribuidas.

6.3.4 Arquitectura de tubería y filtro

El ejemplo final de un patrón arquitectónico es el patrón tubería y filtro (*pipe and filter*). Éste es un modelo de la organización en tiempo de operación de un sistema, donde las transformaciones funcionales procesan sus entradas y producen salidas. Los datos fluyen de uno a otro y se transforman conforme se desplazan a través de la secuencia. Cada paso de procesamiento se implementa como un transformador. Los datos de entrada fluyen por medio de dichos transformadores hasta que se convierten en salida. Las transformaciones pueden ejecutarse secuencialmente o en forma paralela. Es posible que los datos se procesen por cada transformador ítem por ítem o en un solo lote.

El nombre "tubería y filtro" proviene del sistema Unix original, donde era posible vincular procesos empleando "tuberías". Por ellas pasaba una secuencia de texto de un proceso a otro. Los sistemas conformados con este modelo pueden implementarse al combinar comandos Unix, usando las tuberías y las instalaciones de control del intérprete de comandos Unix. Se usa el término "filtro" porque una transformación "filtra" los datos que puede procesar de su secuencia de datos de entrada.

Se han utilizado variantes de este patrón desde que se usaron por primera vez computadoras para el procesamiento automático de datos. Cuando las transformaciones son secuenciales, con datos procesados en lotes, este modelo arquitectónico de tubería y filtro se convierte en un modelo secuencial en lote, una arquitectura común para sistemas de procesamiento de datos (por ejemplo, un sistema de facturación). La arquitectura de un sistema embebido puede organizarse también como un proceso por entubamiento, donde cada proceso se ejecuta de manera concurrente. En el capítulo 20 se estudia el uso de este patrón en sistemas embebidos.

En la figura 6.13 se muestra un ejemplo de este tipo de arquitectura de sistema, que se usa en una aplicación de procesamiento en lote. Una organización emite facturas a los clientes. Una vez a la semana, los pagos efectuados se incorporan a las facturas. Para

Patrones arquitectónicos para control

Existen patrones arquitectónicos específicos que reflejan formas usadas comúnmente para organizar el control en un sistema. En ellos se incluyen el control centralizado, basado en un componente que llama otros componentes, y el control con base en un evento, donde el sistema reacciona a eventos externos.

http://www.SoftwareEngineering-9.com/Web/Architecture/ArchPatterns/

las facturas pagadas se emite un recibo. Para las facturas no saldadas dentro del plazo de pago se emite un recordatorio.

Los sistemas interactivos son difíciles de escribir con el modelo tubería y filtro, debido a la necesidad de procesar una secuencia de datos. Aunque las entradas y salidas textuales simples pueden modelarse de esta forma, las interfaces gráficas de usuario tienen formatos I/O más complejos, así como una estrategia de control que se basa en eventos como clics del mouse o selecciones del menú. Es difícil traducir esto en una forma compatible con el modelo *pipelining* (entubamiento).

6.4 Arquitecturas de aplicación

Los sistemas de aplicación tienen la intención de cubrir las necesidades de una empresa u organización. Todas las empresas tienen mucho en común: necesitan contratar personal, emitir facturas, llevar la contabilidad, etcétera. Las empresas que operan en el mismo sector usan aplicaciones comunes específicas para el sector. De esta forma, además de las funciones empresariales generales, todas las compañías telefónicas necesitan sistemas para conectar llamadas, administrar sus redes y emitir facturas a los clientes, entre otros. En consecuencia, también los sistemas de aplicación que utilizan dichas empresas tienen mucho en común.

Estos factores en común condujeron al desarrollo de arquitecturas de software que describen la estructura y la organización de tipos particulares de sistemas de software. Las arquitecturas de aplicación encapsulan las principales características de una clase de sistemas. Por ejemplo, en los sistemas de tiempo real puede haber modelos arquitectónicos genéricos de diferentes tipos de sistema, tales como sistemas de recolección de datos o sistemas de monitorización. Aunque las instancias de dichos sistemas difieren en detalle, la estructura arquitectónica común puede reutilizarse cuando se desarrollen nuevos sistemas del mismo tipo.

La arquitectura de aplicación puede reimplantarse cuando se desarrollen nuevos sistemas, pero, para diversos sistemas empresariales, la reutilización de aplicaciones es posible sin reimplementación. Esto se observa en el crecimiento de los sistemas de planeación de recursos empresariales (ERP, por las siglas de *Enterprise Resource Planning*) de compañías como SAP y Oracle, y paquetes de software vertical (COTS) para aplicaciones especializadas en diferentes áreas de negocios. En dichos sistemas, un sistema genérico se configura y adapta para crear una aplicación empresarial específica.

Arquitecturas de aplicación

En el sitio Web del libro existen muchos ejemplos de arquitecturas de aplicación. Se incluyen descripciones de sistemas de procesamiento de datos en lote, sistemas de asignación de recursos y sistemas de edición basados en eventos.

http://www.SoftwareEngineering-9.com/Web/Architecture/AppArch/

Por ejemplo, un sistema para suministrar la administración en cadena se adapta a diferentes tipos de proveedores, bienes y arreglos contractuales.

Como diseñador de software, usted puede usar modelos de arquitecturas de aplicación en varias formas:

1. *Como punto de partida para el proceso de diseño arquitectónico* Si no está familiarizado con el tipo de aplicación que desarrolla, podría basar su diseño inicial en una arquitectura de aplicación genérica. Desde luego, ésta tendrá que ser especializada para el sistema específico que se va a desarrollar, pero es un buen comienzo para el diseño.
2. *Como lista de verificación del diseño* Si usted desarrolló un diseño arquitectónico para un sistema de aplicación, puede comparar éste con la arquitectura de aplicación genérica y luego, verificar que su diseño sea consistente con la arquitectura genérica.
3. *Como una forma de organizar el trabajo del equipo de desarrollo* Las arquitecturas de aplicación identifican características estructurales estables de las arquitecturas del sistema y, en muchos casos, es posible desarrollar éstas en paralelo. Puede asignar trabajo a los miembros del grupo para implementar diferentes componentes dentro de la arquitectura.
4. *Como un medio para valorar los componentes a reutilizar* Si tiene componentes por reutilizar, compare éstos con las estructuras genéricas para saber si existen componentes similares en la arquitectura de aplicación.
5. *Como un vocabulario para hablar acerca de los tipos de aplicaciones* Si discute acerca de una aplicación específica o trata de comparar aplicaciones del mismo tipo, entonces puede usar los conceptos identificados en la arquitectura genérica para hablar sobre las aplicaciones.

Hay muchos tipos de sistema de aplicación y, en algunos casos, parecerían muy diferentes. Sin embargo, muchas de estas aplicaciones distintas superficialmente en realidad tienen mucho en común y, por ende, suelen representarse mediante una sola arquitectura de aplicación abstracta. Esto se ilustra aquí al describir las siguientes arquitecturas de dos tipos de aplicación:

1. *Aplicaciones de procesamiento de transacción* Este tipo de aplicaciones son aplicaciones centradas en bases de datos, que procesan los requerimientos del usuario mediante la información y actualizan ésta en una base de datos. Se trata del tipo más común de sistemas empresariales interactivos. Se organizan de tal forma que las acciones del usuario no pueden interferir unas con otras y se mantiene la integridad de la base

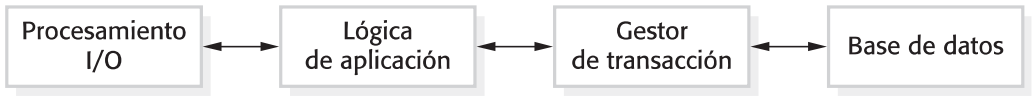


Figura 6.14 Estructura de las aplicaciones de procesamiento de transacción

de datos. Esta clase de sistema incluye los sistemas bancarios interactivos, sistemas de comercio electrónico, sistemas de información y sistemas de reservaciones.

2. *Sistemas de procesamiento de lenguaje* Son sistemas en los que las intenciones del usuario se expresan en un lenguaje formal (como Java). El sistema de procesamiento de lenguaje elabora este lenguaje en un formato interno y después interpreta dicha representación interna. Los sistemas de procesamiento de lenguaje mejor conocidos son los compiladores, que traducen los programas en lenguaje de alto nivel dentro de un código de máquina. Sin embargo, los sistemas de procesamiento de lenguaje se usan también en la interpretación de lenguajes de comandos para bases de datos y sistemas de información, así como de lenguajes de marcado como XML (Harold y Means, 2002; Hunter *et al.*, 2007).

Se eligieron estos tipos particulares de sistemas porque una gran cantidad de sistemas empresariales basados en la Web son sistemas de procesamiento de transacciones, y todo el desarrollo del software se apoya en los sistemas de procesamiento de lenguaje.

6.4.1 Sistemas de procesamiento de transacciones

Los sistemas de procesamiento de transacciones (TP, por las siglas de *Transaction Processing*) están diseñados para procesar peticiones del usuario mediante la información de una base de datos, o los requerimientos para actualizar una base de datos (Lewis *et al.*, 2003). Técnicamente, una transacción de base de datos es una secuencia de operaciones que se trata como una sola unidad (una unidad atómica). Todas las operaciones en una transacción tienen que completarse antes de que sean permanentes los cambios en la base de datos. Esto garantiza que la falla en las operaciones dentro de la transacción no conduzca a inconsistencias en la base de datos.

Desde una perspectiva de usuario, una transacción es cualquier secuencia coherente de operaciones que satisfacen un objetivo, como "encontrar los horarios de vuelos de Londres a París". Si la transacción del usuario no requiere el cambio en la base de datos, entonces sería innecesario empaquetar esto como una transacción técnica de base de datos.

Un ejemplo de una transacción es una petición de cliente para retirar dinero de una cuenta bancaria mediante un cajero automático. Esto incluye obtener detalles de la cuenta del cliente, verificar y modificar el saldo por la cantidad retirada y enviar comandos al cajero automático para entregar el dinero. Hasta que todos estos pasos se completan, la transacción permanece inconclusa y no cambia la base de datos de la cuenta del cliente.

Por lo general, los sistemas de procesamiento de transacción son sistemas interactivos donde los usuarios hacen peticiones asíncronas de servicios. La figura 6.14 ilustra la estructura arquitectónica conceptual de las aplicaciones de TP. Primero, un usuario hace una petición al sistema a través de un componente de procesamiento I/O. La petición se procesa mediante alguna lógica específica de la aplicación. Se crea una transacción y pasa hacia un gestor de transacciones que, por lo general, está embebido en el sistema de manejo de la

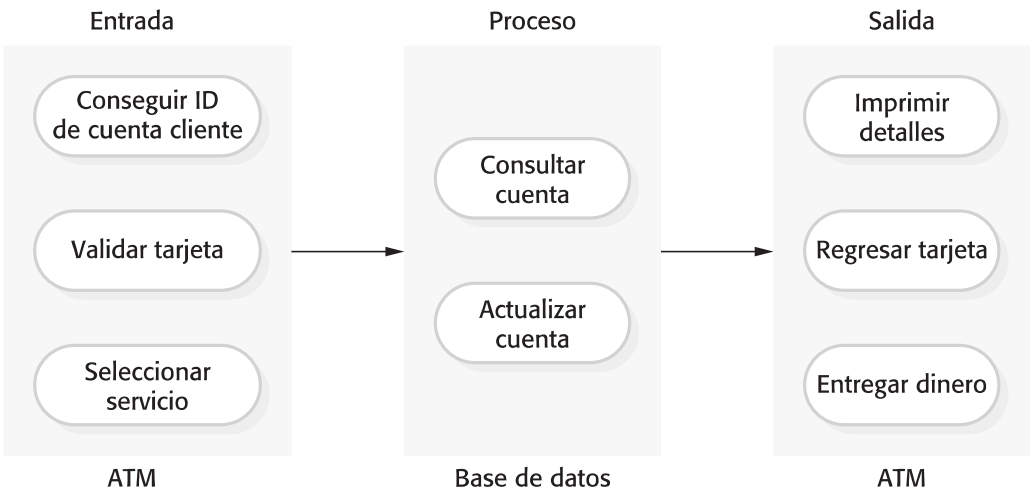


Figura 6.15 Arquitectura de software de un sistema ATM

base de datos. Después de que el gestor de transacciones asegura que la transacción se ha completado adecuadamente, señala a la aplicación que terminó el procesamiento.

Los sistemas de procesamiento de transacción pueden organizarse como una arquitectura "tubería y filtro" con componentes de sistema responsables de entradas, procesamiento y salida. Por ejemplo, considere un sistema bancario que permite a los clientes consultar sus cuentas y retirar dinero de un cajero automático. El sistema está constituido por dos componentes de software cooperadores: el software del cajero automático y el software de procesamiento de cuentas en el servidor de la base de datos del banco. Los componentes de entrada y salida se implementan como software en el cajero automático y el componente de procesamiento es parte del servidor de la base de datos del banco. La figura 6.15 muestra la arquitectura de este sistema e ilustra las funciones de los componentes de entrada, proceso y salida.

6.4.2 Sistemas de información

Todos los sistemas que incluyen interacción con una base de datos compartida se consideran sistemas de información basados en transacciones. Un sistema de información permite acceso controlado a una gran base de información, tales como un catálogo de biblioteca, un horario de vuelos o los registros de pacientes en un hospital. Cada vez más, los sistemas de información son sistemas basados en la Web, cuyo acceso es mediante un navegador Web.

La figura 6.16 presenta un modelo muy general de un sistema de información. El sistema se modela con un enfoque por capas (estudiado en la sección 6.3), donde la capa superior soporta la interfaz de usuario, y la capa inferior es la base de datos del sistema. La capa de comunicaciones con el usuario maneja todas las entradas y salidas de la interfaz de usuario, y la capa de recuperación de información incluye la lógica específica de aplicación para acceder y actualizar la base de datos. Como se verá más adelante, las capas en este modelo pueden trazarse directamente hacia servidores dentro de un sistema basado en Internet.

Como ejemplo de una instancia de este modelo en capas, la figura 6.17 muestra la arquitectura del MHC-PMS. Recuerde que este sistema mantiene y administra detalles de los pacientes que consultan médicos especialistas en problemas de salud mental. En el

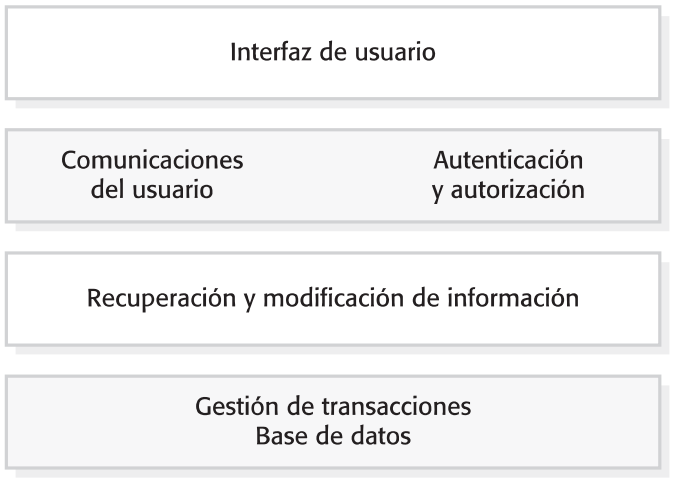


Figura 6.16 Arquitectura de sistema de información en capas

modelo se agregaron detalles a cada capa al identificar los componentes que soportan las comunicaciones del usuario, así como la recuperación y el acceso a la información:

1. La capa superior es responsable de implementar la interfaz de usuario. En este caso, la UI se implementó con el uso de un navegador Web.
2. La segunda capa proporciona la funcionalidad de interfaz de usuario que se entrega a través del navegador Web. Incluye componentes que permiten a los usuarios ingresar al sistema, y componentes de verificación para garantizar que las operaciones utilizadas estén permitidas de acuerdo con su rol. Esta capa incluye componentes de gestión de formato y menú que presentan información a los usuarios, así como componentes de validación de datos que comprueban la consistencia de la información.
3. La tercera capa implementa la funcionalidad del sistema y ofrece componentes que ponen en operación la seguridad del sistema, la creación y actualización de la información del paciente, la importación y exportación de datos del paciente desde otras bases de datos, y los generadores de reporte que elaboran informes administrativos.

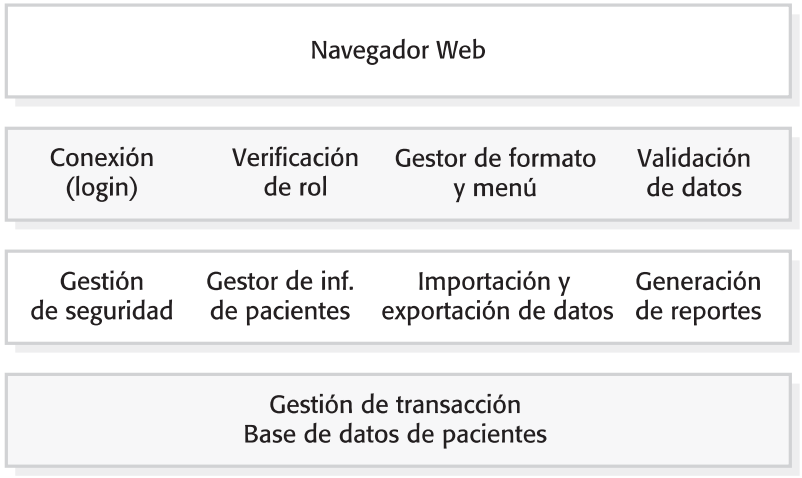


Figura 6.17 Arquitectura del MHC-PMS

4. Finalmente, la capa más baja, que se construye al usar un sistema comercial de gestión de base de datos, ofrece administración de transacciones y almacenamiento constante de datos.

Los sistemas de gestión de información y recursos, por lo general, son ahora sistemas basados en la Web donde las interfaces de usuario se implementan con el uso de un navegador Web. Por ejemplo, los sistemas de comercio electrónico son sistemas de gestión de recursos basados en Internet, que aceptan pedidos electrónicos por bienes o servicios y, luego, ordenan la entrega de dichos bienes o servicios al cliente. En un sistema de comercio electrónico, la capa específica de aplicación incluye funcionalidad adicional que soporta un "carrito de compras", donde los usuarios pueden colocar algunos objetos en transacciones separadas y, luego, pagarlos en una sola transacción.

La organización de servidores en dichos sistemas refleja usualmente el modelo genérico en cuatro capas presentadas en la figura 6.16. Dichos sistemas se suelen implementar como arquitecturas cliente-servidor de multinivel, como se estudia en el capítulo 18:

1. El servidor Web es responsable de todas las comunicaciones del usuario, y la interfaz de usuario se pone en función mediante un navegador Web;
2. El servidor de aplicación es responsable de implementar la lógica específica de la aplicación, así como del almacenamiento de la información y las peticiones de recuperación;
3. El servidor de la base de datos mueve la información hacia y desde la base de datos y, además, manipula la gestión de transacciones.

El uso de múltiples servidores permite un rendimiento elevado, al igual que posibilita la manipulación de cientos de transacciones por minuto. Conforme aumenta la demanda, pueden agregarse servidores en cada nivel, para lidiar con el procesamiento adicional implicado.

6.4.3 Sistemas de procesamiento de lenguaje

Los sistemas de procesamiento de lenguaje convierten un lenguaje natural o artificial en otra representación del lenguaje y, para lenguajes de programación, también pueden ejecutar el código resultante. En ingeniería de software, los compiladores traducen un lenguaje de programación artificial en código de máquina. Otros sistemas de procesamiento de lenguaje traducen una descripción de datos XML en comandos para consultar una base de datos o una representación XML alternativa. Los sistemas de procesamiento de lenguaje natural pueden transformar un lenguaje natural a otro, por ejemplo, francés a noruego.

En la figura 6.18 se ilustra una posible arquitectura para un sistema de procesamiento de lenguaje hacia un lenguaje de programación. Las instrucciones en el lenguaje fuente definen el programa a ejecutar, en tanto que un traductor las convierte en instrucciones para una máquina abstracta. Luego, dichas instrucciones se interpretan mediante otro componente que lee (*fetch*) las instrucciones para su ejecución y las ejecuta al usar (si es necesario) datos del entorno. La salida del proceso es el resultado de la interpretación de instrucciones en los datos de entrada.

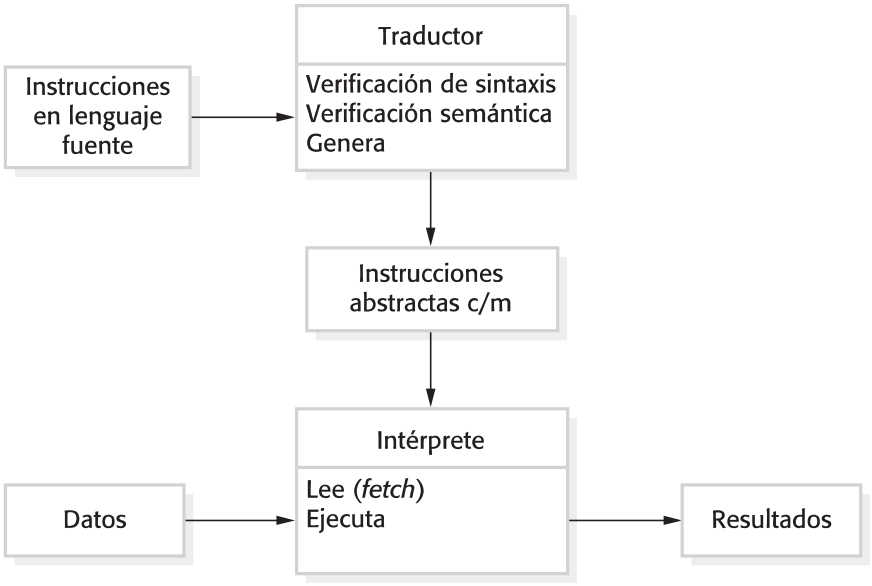


Figura 6.18 Arquitectura de un sistema de procesamiento de lenguaje

Desde luego, para muchos compiladores, el intérprete es una unidad de hardware que procesa instrucciones de la máquina, en tanto que la máquina abstracta es el procesador real. Sin embargo, para lenguajes escritos de manera dinámica, como Python, el intérprete puede ser un componente de software.

Los compiladores de lenguaje de programación que forman parte de un entorno de programación más general tienen una arquitectura genérica (figura 6.19) que incluye los siguientes componentes:

1. Un analizador léxico, que toma valores simbólicos (*tokens*) y los convierte en una forma interna.
2. Una tabla de símbolos, que contiene información de los nombres de las entidades (variables, de clase, de objeto, etcétera) usados en el texto que se traduce.
3. Un analizador de sintaxis, el cual verifica la sintaxis del lenguaje que se va a traducir. Emplea una gramática definida del lenguaje y construye un árbol de sintaxis.
4. Un árbol de sintaxis es una estructura interna que representa el programa a compilar.

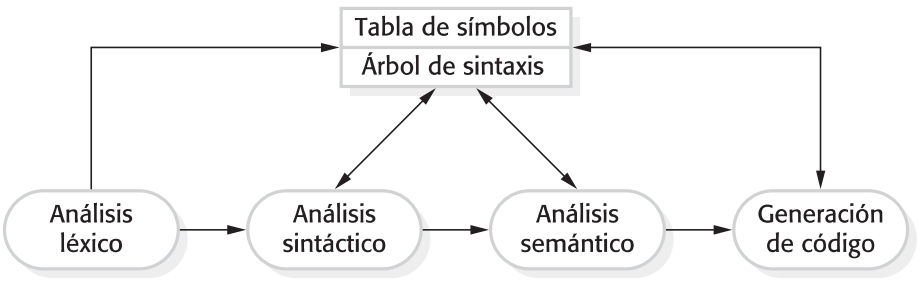


Figura 6.19 Una arquitectura de compilador de tubería y filtro

Arquitecturas de referencia

Las arquitecturas de referencia captan en un dominio características importantes de las arquitecturas del sistema. En esencia, incluyen todo lo que pueda estar en una arquitectura de aplicación aunque, en realidad, es muy improbable que alguna aplicación individual contenga todas las características mostradas en una arquitectura de referencia. El objetivo principal de las arquitecturas de referencia consiste en evaluar y comparar las propuestas de diseño y, en dicho dominio, educar a las personas sobre las características arquitectónicas.

http://www.SoftwareEngineering-9.com/Web/Architecture/RefArch.html

5. Un analizador semántico que usa información del árbol de sintaxis y la tabla de símbolos, para verificar la exactitud semántica del texto en lenguaje de entrada.
6. Un generador de código que "recorre" el árbol de sintaxis y genera un código de máquina abstracto.

También pueden incluirse otros componentes que analizan y transforman el árbol de sintaxis para mejorar la eficiencia y remover redundancia del código de máquina generado. En otros tipos de sistema de procesamiento de lenguaje, como un traductor de lenguaje natural, habrá componentes adicionales, por ejemplo, un diccionario, y el código generado en realidad es el texto de entrada traducido en otro lenguaje.

Existen patrones arquitectónicos alternativos que pueden usarse en un sistema de procesamiento de lenguaje (Garlan y Shaw, 1993). Pueden implementarse compiladores con una composición de un repositorio y un modelo de tubería y filtro. En una arquitectura de compilador, la tabla de símbolos es un repositorio para datos compartidos. Las fases de análisis léxico, sintáctico y semántico se organizan de manera secuencial, como se muestra en la figura 6.19, y se comunican a través de la tabla de símbolos compartida.

Este modelo de tubería y filtro de compilación de lenguaje es efectivo en entornos *batch*, donde los programas se compilan y ejecutan sin interacción del usuario; por ejemplo, en la traducción de un documento XML a otro. Es menos efectivo cuando un compilador se integra con otras herramientas de procesamiento de lenguaje, como un sistema de edición estructurado, un depurador interactivo o un programa de impresión estética (*prettyprinter*). En esta situación, los cambios de un componente deben reflejarse de inmediato en otros componentes. Por lo tanto, es mejor organizar el sistema en torno a un repositorio, como se muestra en la figura 6.20.

Esta figura ilustra cómo un sistema de procesamiento de lenguaje puede formar parte de un conjunto integrado de herramientas de soporte de programación. En este ejemplo, la tabla de símbolos y el árbol de sintaxis actúan como un almacén de información central. Las herramientas y los fragmentos de herramienta se comunican a través de él. Otra información que en ocasiones se incrusta en las herramientas, como la definición gramática y la definición del formato de salida para el programa, se toma de las herramientas y se coloca en el repositorio. En consecuencia, un editor enfocado en la sintaxis podría verificar que ésta sea correcta mientras se escribe un programa, y un *prettyprinter* puede crear listados del programa en un formato que sea fácil de leer.

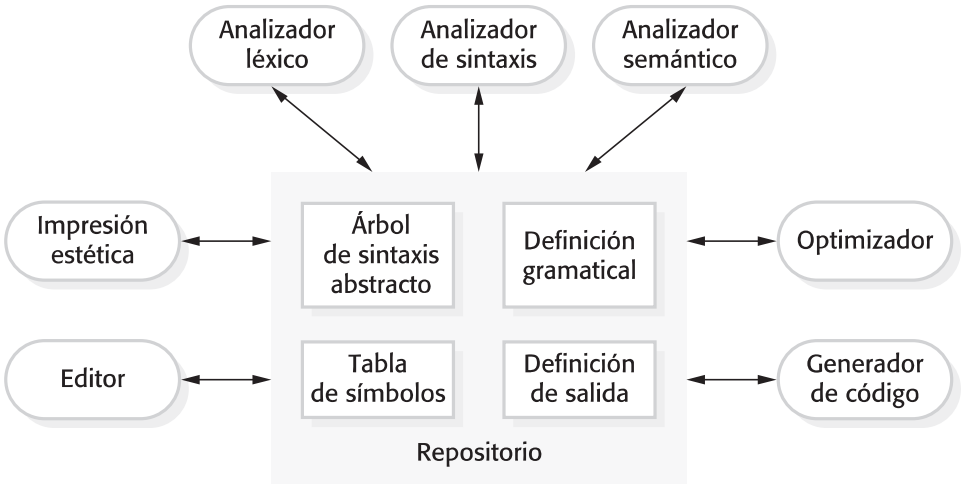


Figura 6.20 Arquitectura de repositorio para un sistema de procesamiento de lenguaje

PUNTOS CLAVE

- Una arquitectura de software es una descripción de cómo se organiza un sistema de software. Las propiedades de un sistema, como rendimiento, seguridad y disponibilidad, están influidas por la arquitectura utilizada.
- Las decisiones de diseño arquitectónico incluyen decisiones sobre el tipo de aplicación, la distribución del sistema, los estilos arquitectónicos a usar y las formas en que la arquitectura debe documentarse y evaluarse.
- Las arquitecturas pueden documentarse desde varias perspectivas o diferentes vistas. Las posibles vistas incluyen la conceptual, la lógica, la de proceso, la de desarrollo y la física.
- Los patrones arquitectónicos son medios para reutilizar el conocimiento sobre las arquitecturas de sistemas genéricos. Describen la arquitectura, explican cuándo debe usarse, y exponen sus ventajas y desventajas.
- Los patrones arquitectónicos usados comúnmente incluyen el modelo de vista del controlador, arquitectura en capas, repositorio, cliente-servidor, y tubería y filtro.
- Los modelos genéricos de las arquitecturas de sistemas de aplicación ayudan a entender la operación de las aplicaciones, comparar aplicaciones del mismo tipo, validar diseños del sistema de aplicación y valorar componentes para reutilización a gran escala.
- Los sistemas de procesamiento de transacción son sistemas interactivos que permiten el acceso y la modificación remota de la información, en una base de datos por parte de varios usuarios. Los sistemas de información y los sistemas de gestión de recursos son ejemplos de sistemas de procesamiento de transacciones.
- Los sistemas de procesamiento de lenguaje se usan para traducir textos de un lenguaje a otro y para realizar las instrucciones especificadas en el lenguaje de entrada. Incluyen un traductor y una máquina abstracta que ejecuta el lenguaje generado.

LECTURAS SUGERIDAS

Software Architecture: Perspectives on an Emerging Discipline. Éste fue el primer libro sobre arquitectura de software e incluye una amplia discusión acerca de los diferentes estilos arquitectónicos. (M. Shaw y D. Garlan, Prentice-Hall, 1996.)

Software Architecture in Practice, 2nd ed. Se trata de una cuestión práctica de arquitecturas de software que no exagera los beneficios del diseño arquitectónico. Ofrece una clara razón empresarial sobre por qué son importantes las arquitecturas. (L. Bass, P. Clements y R. Kazman, Addison-Wesley, 2003.)

"The Golden Age of Software Architecture". Este ensayo estudia el desarrollo de la arquitectura de software, desde sus inicios en la década de 1980 hasta su uso actual. Aun cuando tiene poco contenido técnico, presenta un panorama histórico amplio e interesante. (M. Shaw y P. Clements, *IEEE Software,* **21** (2), marzo-abril 2006.) http://dx.doi.org/10.1109/MS.2006.58.

Handbook of Software Architecture. Éste es un trabajo en progreso de Grady Booch, uno de los primeros difusores de la arquitectura de software. Ha documentado las arquitecturas de varios sistemas de software, de manera que el lector puede ver la realidad en vez de abstracción académica. Disponible en la Web y con la intención de aparecer como libro. http://www.handbookofsoftwarearchitecture.com/.

EJERCICIOS

6.1. Cuando se describe un sistema, explique por qué es posible que deba diseñar la arquitectura del sistema antes de completar la especificación de requerimientos.

6.2. Se le pide preparar y entregar una presentación a un administrador no técnico para justificar la contratación de un arquitecto de sistemas para un nuevo proyecto. Escriba una lista que establezca los puntos clave de su presentación. Por supuesto, debe explicar qué se entiende por arquitecto de sistemas.

6.3. Exponga por qué pueden surgir conflictos de diseño cuando se desarrolla una arquitectura para la que tanto los requerimientos de disponibilidad como los de seguridad son los requerimientos no funcionales más importantes.

6.4. Dibuje diagramas que muestren una vista conceptual y una vista de proceso de las arquitecturas de los siguientes sistemas:

Un sistema automatizado de emisión de boletos que utilizan los pasajeros en una estación de ferrocarril.

Un sistema de videoconferencia controlado por computadora, que permita que los datos de video, audio y computadora sean al mismo tiempo visibles a muchos participantes.

Un robot limpiador de pisos cuya función sea asear espacios relativamente despejados, como corredores. El limpiador debe detectar las paredes y otros obstáculos.

Arquitectura en capas

Capa HTML:

Mi Portafolio

Inicio Sobre mí Contacto

Bienvenido

Esta es mi primera página aplicando diseño arquitectónico con HTML y CSS.

© 2025 Todos los derechos reservados

Capa CSS:

Estilo general del documento (body)

- Toda la página tendrá letra Arial (si no está disponible, se usará una fuente sin serif).
- El fondo será de color gris claro #f4f6f9.
- Se eliminan los márgenes y espacios por defecto del navegador.

Encabezado (header)

- El encabezado tendrá fondo azul oscuro #2c3e50.
- El texto será blanco y estará centrado.
- Se agregará un espacio de 15px de relleno alrededor del contenido.

Menú de navegación (nav a)

- Los enlaces dentro del menú de navegación serán de color blanco.
- Tendrán un espacio de 15px a la izquierda y derecha para separarlos.
- No tendrán subrayado (text-decoration: none).

Contenido principal (main)

- El contenido principal de la página tendrá un margen interno de 20px para que el texto no quede pegado a los bordes.

Pie de página (footer)

- El pie de página tendrá fondo gris #bdc3c7.
- El texto será centrado y más pequeño (12px).
- Se agregará un relleno de 10px.

Cliente - Servidor

Cliente

Servidor

Respuesta del Servidor

✅ El servidor ha recibido tu mensaje. ¡Gracias por contactarnos!

Capa HTML del Cliente:

Formulario de Contacto

Nombre: Correo: Mensaje: Enviar

© 2025 Cliente–Servidor

Capa HTML del Servidor:

Respuesta del Servidor

✅ El servidor ha recibido tu mensaje. ¡Gracias por contactarnos!

Capa CSS:

Estilo general del documento (body)

- Toda la página usará la fuente Arial (si no está disponible, se aplica una fuente sans-serif).
- El fondo será de color gris claro #f4f6f9.
- Se eliminan los márgenes y espacios que el navegador aplica por defecto.

Encabezado (header)

- El encabezado tendrá un fondo azul oscuro #2c3e50.
- El texto será blanco y centrado.
- Se añade un espacio interno (padding) de 15px

Formulario (form)

- El formulario estará centrado en la página (margin: auto).
- Tendrá un ancho máximo de 400px.
- Fondo blanco, bordes redondeados (border-radius) y una ligera sombra (box-shadow) para destacarlo.

Etiquetas (label)

- Cada etiqueta se muestra en una línea completa (block).
- Se deja espacio entre el texto de la etiqueta y los campos.

Campos del formulario (input, textarea, button)

- Todos los campos ocupan el 100% del ancho disponible.
- Tienen un relleno interno de 8px y un margen inferior de 10px para separarlos.

Botones (button)

- El botón tendrá fondo azul #2980b9, texto blanco y sin borde.
- Al pasar el mouse por encima (hover), el fondo cambia a un azul más claro.
- El cursor cambia a la mano apuntando.

Pie de página (footer)

- El pie de página tendrá fondo gris claro #bdc3c7.
- El texto estará centrado y en tamaño pequeño (12px).
- Se aplica un espacio interno de 10px.