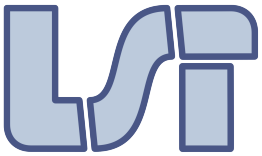
Fundamentos de Ingeniería del Software

grado en ingeniería informática

Resumen del temario

Autor
Carlos Sánchez Páez



Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Curso 2017-2018

Índice

Índice de figuras

1. Tema 1. Introducción a la Ingeniería del Software

1.1. El producto Software

1.1.1. Definición de Software

El software se puede definir de varias formas:

- Programa o conjunto de programas de cómputo que incluye datos, procedimientos y pautas que permiten realizar distintas tareas en un sistema informático.
- Transformador de información, para lo que adquiere, gestiona, modifica, produce o transmite esa información.

1.1.2. Tipos de software

Podemos clasificar el software en varios tipos:

1. Por *campo de aplicación*:
   a) Software de **sistemas**.
   b) Software de **aplicaciones**.
   c) Sotware de **programación**.
2. Por *tipo de licencia*:
   a) Según derechos de autor
      1) Software de **código abierto**.
      2) Software de **código cerrado**.
      3) Software de **dominio público**.
   b) Según su destinatario
      1) Usuario final (software **hecho a medida**).
      2) Para distribución (software **genérico**).

1.1.3. Características principales

1. El software es un producto lógico:
   a) No se fabrica, sino que se desarrolla.
   b) No se estropea, sino que se deteriora.
2. Crea modelos de la realidad.
3. Está formado por múltiples piezas que deben encajar perfectamente.

1.1.4. Proceso de producción

El proceso de producción del software está formado por las siguientes etapas:

1. **Definición**. Debemos precisar lo que queremos desarrollar. Para ello debemos realizar varias tareas:
   - Ingeniería de Sistemas.
   - Ingeniería de Requisitos.
   - Planificación de proyectos.
2. **Construcción**. Hemos de determinar cómo desarrollaremos el software. Depende de:
   - Diseño del software.
   - Generación del código.
   - Prueba del software.
3. **Evolución**. Consiste en precisar las partes del software que cambiarán.
   - Corrección.
   - Adaptación.
   - Mejora.
   - Prevención.

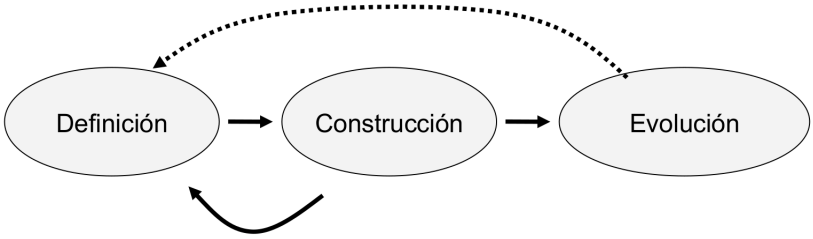
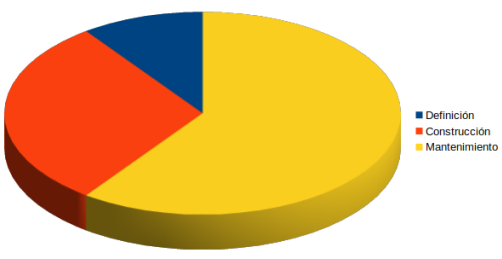
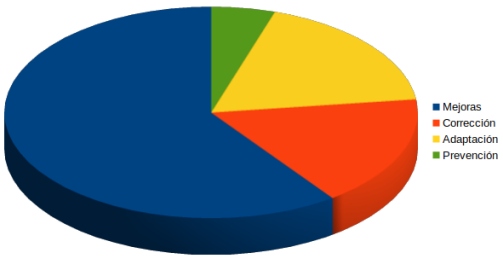


Figura 1: Etapas del proceso de producción del software.



(a) Esfuerzo invertido en las distintas etapas.



(b) Esfuerzo invertido en las distintas etapas del mantenimiento.

Figura 2: Comparativa de esfuerzos.

1.1.5. Problemas en el desarrollo

1. **Comunicación** entre personas.

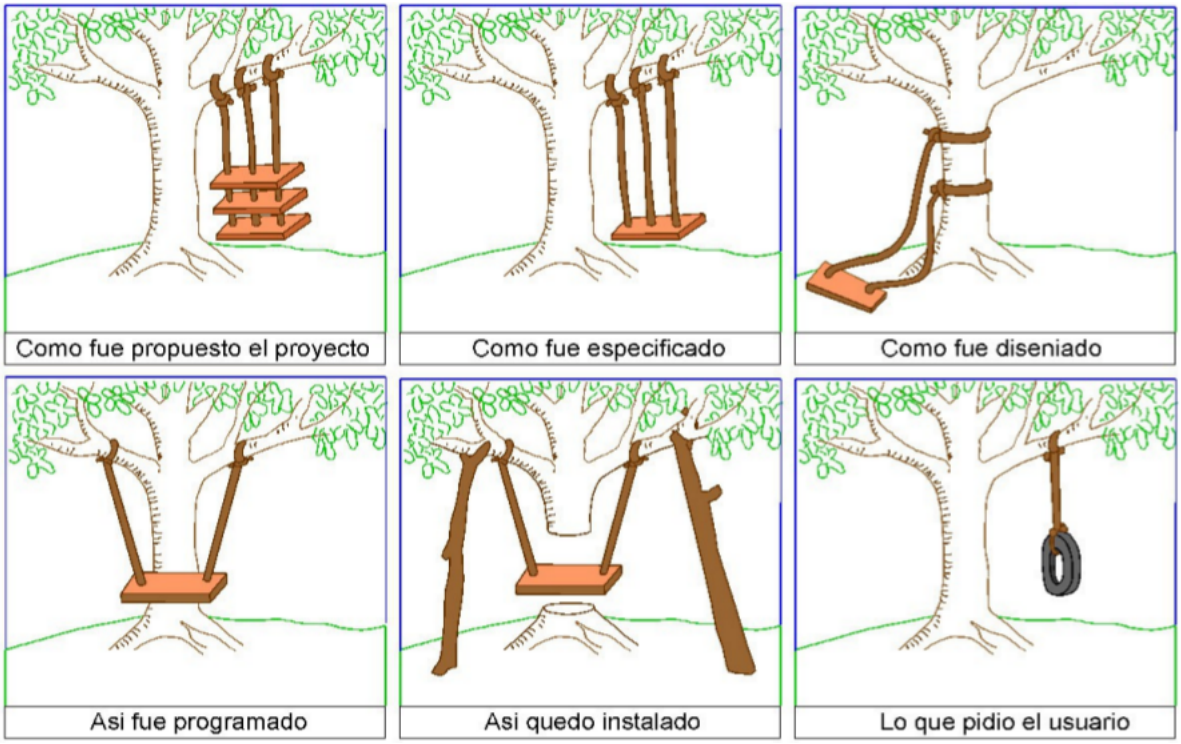


Figura 3: Problemas en la comunicación.

2. Incumplimiento de la **planificación**.
3. Incorporación de **cambios** en etapas avanzadas.

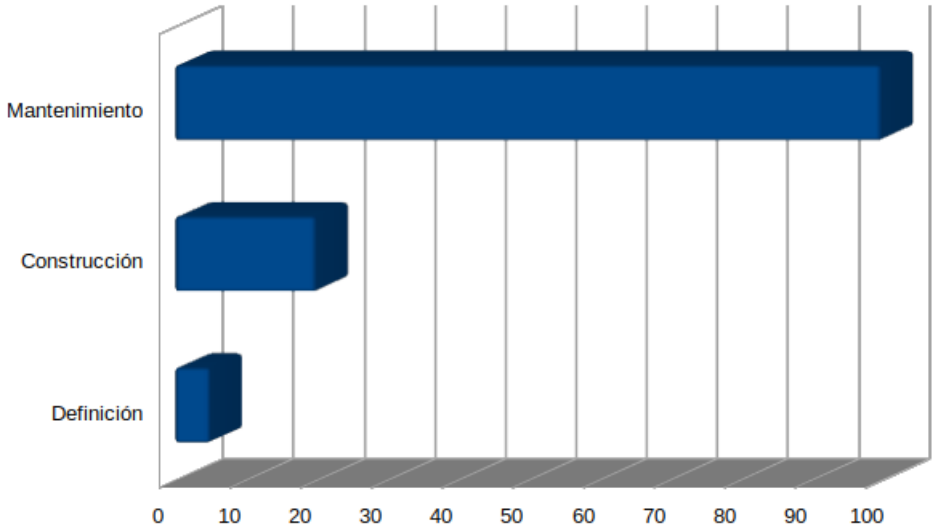


Figura 4: Impacto del cambio en las distintas etapas.

Desastres ocasionados por sistemas software La mayoría de ellos tienen como causa pruebas deficientes, mala documentación, diseños pobres o inexistentes, mal estudio del problema, etc.

1.2. Concepto de Ingeniería del Software

La *Ingeniería del Software* surgió por varias necesidades:

- Mal funcionamiento (calidad).
- Mantenimiento del software existente.
- Demanda creciente del nuevo software.
- Adaptación a las nuevas tecnologías.
- Incremento de la complejidad.

1.2.1. Definiciones de la Ingeniería del Software

1. Establecimiento de los principios y métodos de la ingeniería a fin de obtener software de modo rentable que sea fiable y trabaje en máquinas reales.
2. Aplicación práctica del conocimiento científico en el diseño y construcción de programas de computadora y la documentación asociada y requerida para el desarrollo, operación y mantenimiento del programa.
3. Estudio de los principios y métodos para el desarrollo y mantenimiento de sistemas software.
4. Aplicación de un enfoque sistémico, disciplinado y cuantificable al desarrollo, operación y mantenimiento del software; es decir, aplicación de la ingeniería al software.
5. Conjunto de teorías, métodos e instrumentos (tecnológicos y organizativos) que permitan construir sistemas software con las características de calidad deseadas.
6. Disciplina de ingeniería que se interesa por todos los aspectos de la producción de software, desde las primerasa etapas de la especificación hasta el mantenimiento del sistema después de su puesta en operación.

1.2.2. Terminología usada en Ingeniería del Software

- **Sistema**. Conjunto de elementos relacionados entre sí y con el medio, que forman una unidad o un todo organizativo.
- **Sistema basado en computadora**. Conjunto o disposición de elementos organizados para cumplir una meta predefinida al procesar información.

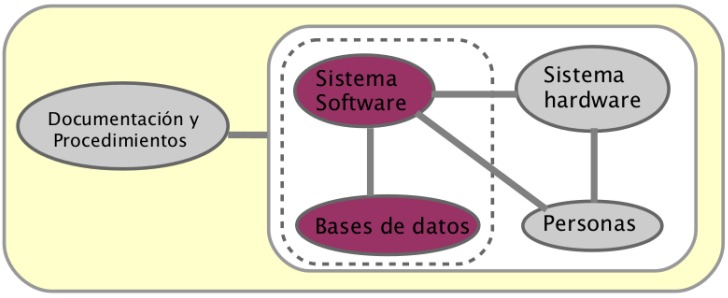


Figura 5: Sistema basado en computadora.

- **Sistema Software**. Conjunto de piezas o elementos software relacionados entre sí y organizados en subsistemas.

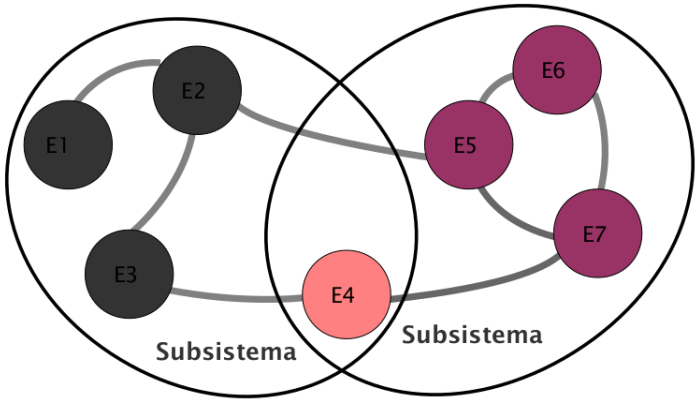


Figura 6: Sistema Software.

- **Modelo**. Representación de un problema en un determinado lenguaje. De un mismo problema se pueden construir muchos modelos.

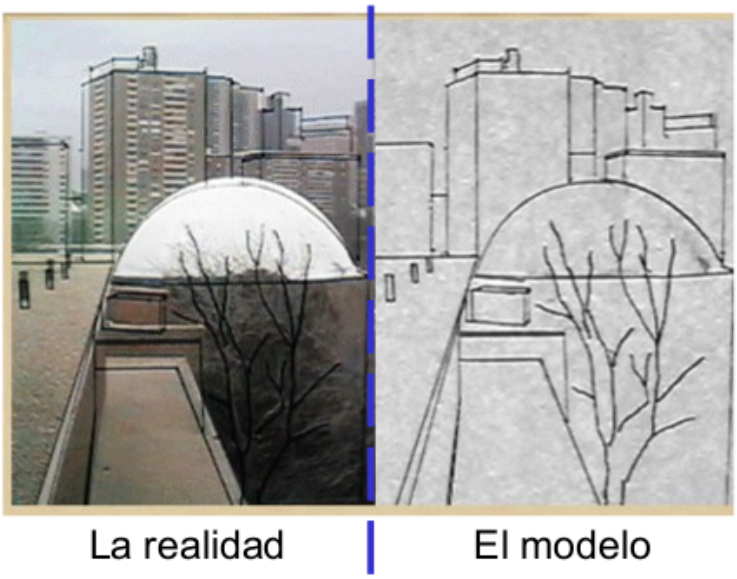


Figura 7: Modelo.

- **Principio**. Elementos que son adquiridos mediante el conocimiento. Determinan las características que debe poseer un modelo para ser una representación adecuada de un sistema.
- **Herramienta**. Instrumentos que permiten la representación de modelos.
- **Técnica**. Modo de utilización de las herramientas.
- **Heurísticas**. Conjunto de reglas empíricas que al ser aplicadas producen modelos que se adecuan a los principios. Ejemplo: *No usar materiales flexibles para representar la maqueta de un edificio.*

- **Proceso**. Estructura que debe establecerse para la obtención eficaz de un producto de Ingeniería.
- **Método**. Proporciona la experiencia técnica para elaborar el producto software. Se basa en principios fundamentales e incluye actividades de modelado.

1.3. Proceso de desarrollo del software

1.3.1. Concepto de proceso de desarrollo

- **Proceso de desarrollo del software**. Conjunto de *actividades, acciones y tareas* que se realizan cuando va a crearse un producto o sistema software.
- **Actividad**. Busca el logro de objetivos amplios e independientes del tipo de aplicación a desarrollar y su complejidad.
- **Acción**. Conjunto de tareas que elaboran un producto importante como resultado.
- **Tarea**. Objetivo pequeño y bien definido que produce un resultado tangible.

Las actividades que se realizan pueden ser de varios tipos:

1. **Estructurales**. Se dedican a obtener el producto:
   a) **Comunicación**. Colaboración con el cliente para entender los objetivos y requisitos del proyecto.
   b) **Planificación**. Definición del plan de proyecto en el que se describen los riesgos probables, recursos adquiridos y productos obtenidos a la vez que se programan las actividades, acciones y tareas.
   c) **Modelado**. Representación mediante modelos del sistema porpuesto junto con la solución o soluciones apropiadas.
   d) **Construcción**. Generación de código y su prueba.
   e) **Despliegue**. Entrega al consumidor y evaluación por parte de éste, lo que sirve como retroalimentación para el equipo de desarrollo.
2. **Sombrilla**. Se aplican a lo largo de todo el proceso. Se dedican a:
   a) Seguimiento y control del proyecto.
   b) Administración del riesgo.
   c) Aseguramiento de la calidad.
   d) Revisiones técnicas.
   e) Mediciones de parámetros del proceso.
   f) Administración de la configuración.
   g) Administración de la reutilización.
   h) Preparación y producción del producto de trabajo.

1.3.2. Modelo general de proceso

- **Estructura del proceso**. Cada una de las actividades, acciones y tareas se encuadran dentro de una estructura que define su relación con el proceso y entre ellas.

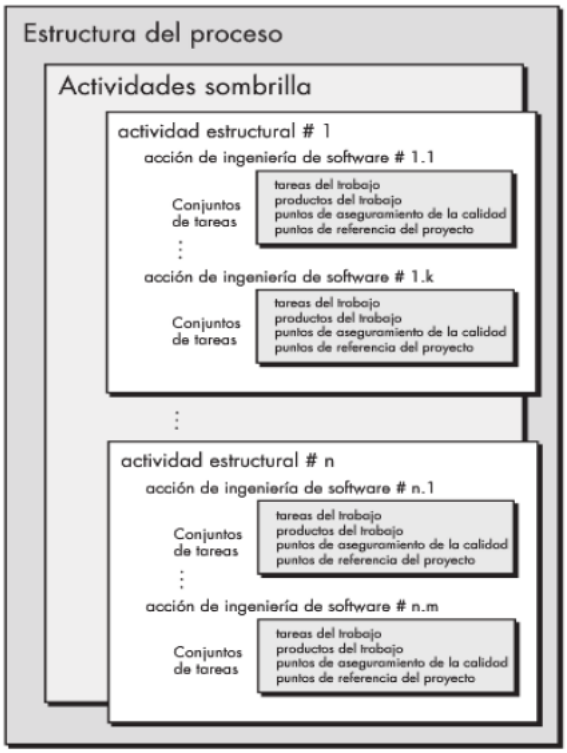
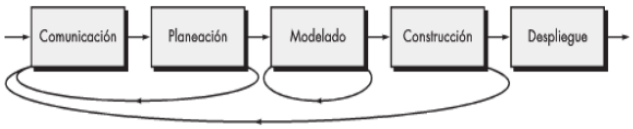


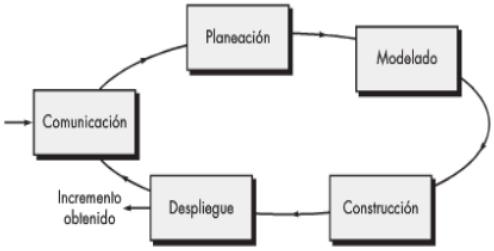
Figura 8: Estructura del proceso.

- **Flujo del proceso**. Describe la forma en la que se organizan las actividades estructurales, acciones y tareas en los procesos con respecto a la secuencia y el tiempo.

(a) Flujo de proceso lineal.



(b) Flujo de proceso iterativo.



(c) Flujo de proceso evolutivo.

Figura 9: Tipos de flujos de proceso.

- **Acciones y tareas de las actividades estructurales**.

  1. **Obtención de requisitos**. Obtención de información respecto a la acción que debe realizar el software.
  2. **Estimación y planificación del proyecto**. Estimar el tiempo y los costes del desarrollo del software.
  3. **Análisis de requisitos**. Documento en el que se especifica lo que debe hacer el sistema software.
  4. **Diseño**. Búsqueda de la solución. Descripción de los componentes, sus relaciones y funciones que le dan solución al problema.
  5. **Implementación**. Traducción del diseño a un lenguaje de programación entendible por una máquina.
  6. **Prueba del software**. Revisión y validación del código que se va desarrollando.
  7. **Evaluación y aceptación**. Evaluación del producto y aceptación por parte de los interesados en él.
  8. **Entrega y asistencia**. Sistema pasa a operar y se ofrece asistencia para su correcto funcionamiento.

1.3.3. Tipos de modelos de proceso

- **Modelo en cascada**. Presenta una estructura secuencial y un flujo lineal. Sin embargo, los proyectos no suelen adecuarse a este modelo, es difícil expresar los requisitos a través de él al principio del proyecto y ofrece poca comunicación con el cliente, ya que hasta el final no hay un ejecutable que se pueda evaluar.

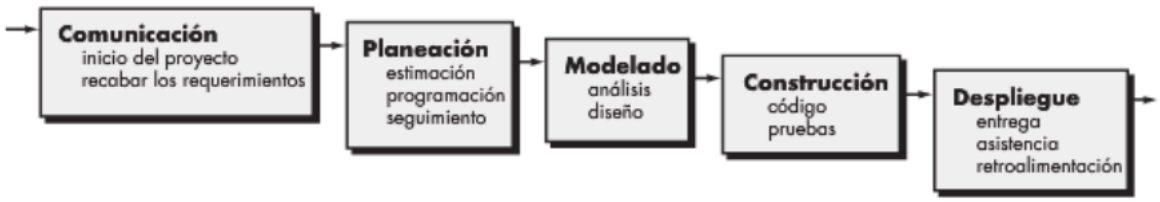


Figura 10: Modelo en cascada.

- **Modelo incremental**. Su estructura es secuencial mientras que el flujo de proceso es lineal y paralelo entre incrementos.

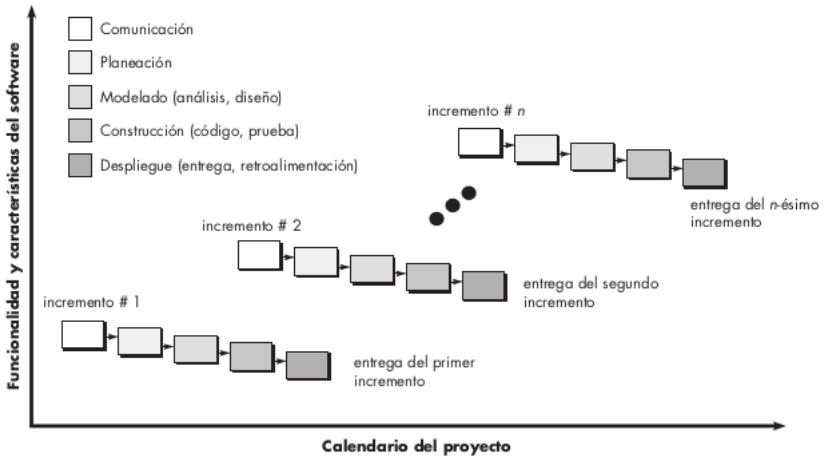


Figura 11: Modelo incremental.

- **Modelo evolutivo**. Es también iterativo. Nace como solución a varios factores, como un tiempo de entrega muy limitado o la necesidad de facilitar la incorporación de cambios. En cada iteración del proceso se obtiene un producto terminado y operativo. Sus características generales son:

  1. Afrontan los riesgos altos (técnicos, de requisitos, etc.) tan pronto como sea posible.
  2. Retroalimentación temprana por parte del cliente.
  3. Manejo de la complejidad (pasos cortos y sencillos).
  4. El conocimiento adquirido durante una iteración de la evolución se puede usar en el resto de iteraciones.
  5. Involucra continuamente al usuario (evaluación, retroalimentación, afinamiento y refinamiento de requisitos, etc.).

Hay dos tipos fundamentales de modelos evolutivos:

1. **Modelo de prototipos**. Un prototipo es una representación limitada de un producto que se utiliza para probar su diseño y comprender mejor el problema y sus posibles soluciones. Los prototipos pueden ser *evolutivos* (productos finales) o *desechables* (usados dentro de otros modelos de proceso).

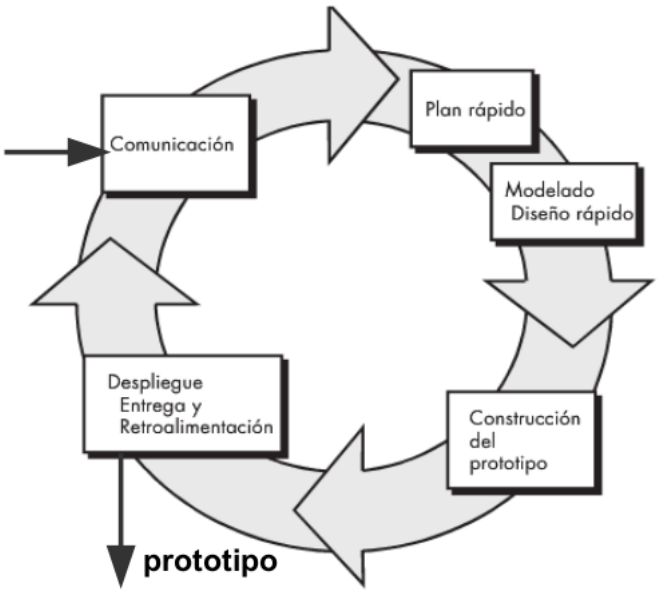


Figura 12: Modelo de prototipos

Este modelo se utiliza para:

1. Facilitar la obtención y validación de requisitos (*desechable*).
2. Estudios de viabilidad (*desechable*).
3. Propuestas de diseños alternativos (*desechable*).
4. En casos muy concretos como producto final (*evolutivo*).

Presentan todas las características de los modelos evolutivos, aunque se les añaden algunos inconvenientes:

- Crear falsas expectativas por parte del cliente (*desechable*).
- Puede que el prototipo *desechable* se elabore con una metodología ineficiente y ésta se mantenga en el producto final.

- **Modelo en espiral de Boehm**. Además de las características de los procesos iterativos, incluye otras más:

1. Se centra en el análisis de riesgo, construyendo prototipos para su estudio.
2. La espiral puede continuar una vez que se entregue el momento para llevar a cabo el mantenimiento.
3. Es adecuado para el desarrollo de sistemas a gran escala.

Sus principales inconvenientes son que no es controlable y que requiere un equipo de desarrollo con gran experiencia en análisis de riesgo.

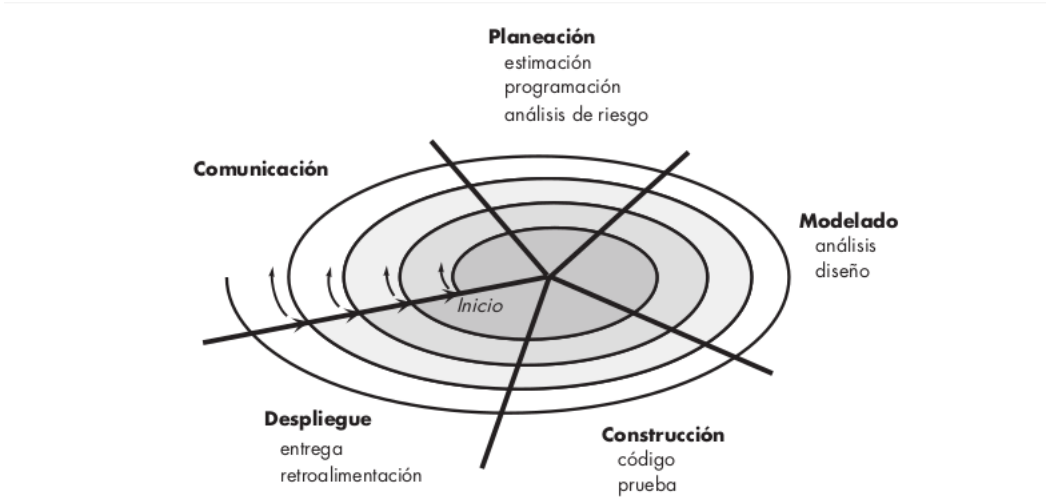


Figura 13: Modelo en espiral de Boehm

1.3.4. Proceso unificado

Es un modelo de proceso evolutivo y compuesto por cuatro fases:

- Inicio o concepción.
- Elaboración.
- Construcción.
- Transición.

Estas etapas se reparten entre las actividades estructurales como podemos ver en el diagrama:

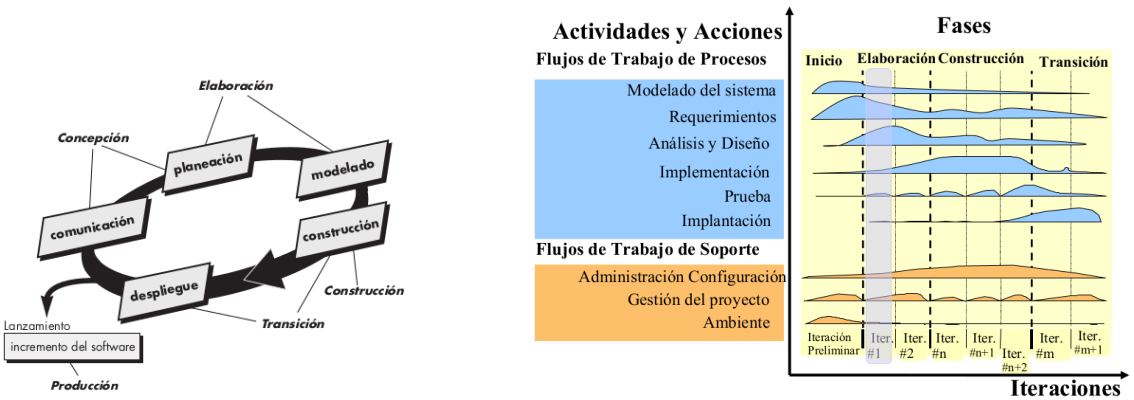


Figura 14: Proceso unificado.

Además de las características de los modelos de proceso evolutivos, el proceso unificado incorpora las siguientes:

1. Es un modelo de proceso adaptable a la complejidad y al tipo de sistema.
2. Está centrado en la arquitectura, mostrando y decidiendo los distintos aspectos arquitectónicos de un sistema software en etapas tempranas. De esta forma pueden servir de base a las posteriores.
3. Está dirigido por casos de uso, desarrollándose uno o varios en cada iteración. En iteraciones tempranas los casos de uso determinarán la arquitectura.

Acciones y tareas en cada fase

- **Inicio**. Agrupa actividades tanto de comunicación del cliente como de planificación. Se propone una arquitectura aproximada para el sistema y se estudian numerosos factores (viabilidad, alcance, riesgos, etc.).
- **Elaboración**. Incluye actividades de comunicación y modelado de la arquitectura básica sobre la que se asentará la fase de construcción.
- **Construcción**. Se completan los modelos de requisitos y se implementan los elementos necesarios para completar el sistema. Según se van terminando los elementos, son probados e integrados al producto final. También se realizan pruebas de aceptación por parte del usuario.
- **Transición**. Consiste en asegurarse de que el sistema cumple con los requisitos especificados (mediante pruebas por parte de los usuarios). Además, se genera el material necesario para lanzar el producto al mercado.

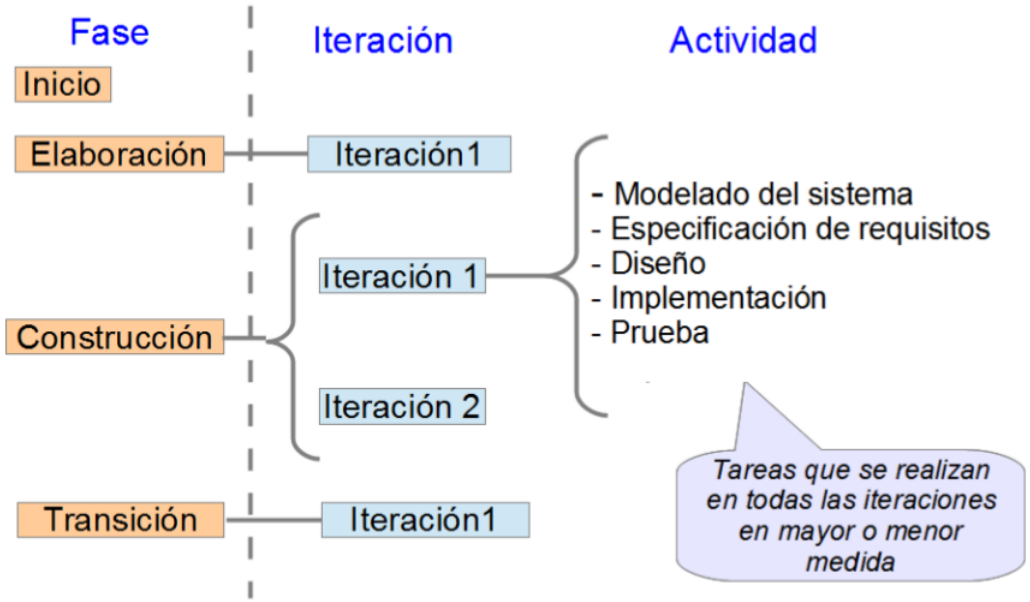


Figura 15: Ejemplo de Proceso Unificado

1.3.5. Desarrollo Ágil

En 2001, diecisiete expertos se preguntaron por qué muchos proyectos generaban menos valor del esperado, no se terminaban a tiempo, tenían problemas de calidad serios, etc. Tras ésto, elaboraron el *Manifiesto el Desarrollo Ágil de Software* con el que intentaban dar solución a estos problemas.

Características del Desarrollo Ágil

1. Es un proceso iterativo e incremental, por lo que es evolutivo.
2. Requiere entregas frecuentes y trabajo en equipo.
3. Establece autonomía en el equipo de desarrollo.
4. Exige revisiones y reuniones retrospectivas frecuentes.

Sus beneficios son que se mejora la productividad y que se manejan mejor los riesgos.
Consta de varias técnicas:

- Scrum
- XP (*Extreme Programming*)
- Programación en parejas.
- TDD (*Test Driven Development*)

2. Tema 2. Ingeniería de Requisitos

2.1. Introducción

En 1995 se realizó el informe *CHAOS* sobre los resultados que se obtuvieron en diversos proyectos software:

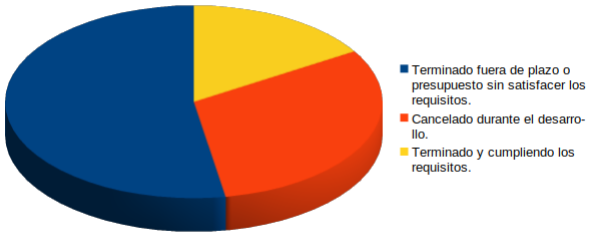


Figura 16: Resultados del informe *CHAOS*.

Los principales factores de fracaso son:

1. Falta de información por parte de los usuarios.
2. Especificación de requisitos incompleta.
3. Continuos cambios de los requisitos.
4. Pobres habilidades técnicas en la especificación de requisitos.

La **Ingeniería de Requisitos** cubre las tareas y proporciona los mecanismos adecuados para:

- Entender y analizar las necesidades del cliente.
- Evaluar la viabilidad de las necesidades.
- Negociar una solución razonable.
- Especificar la solución sin ambigüedades, confeccionando un documento que describa la solución acordada.
- Validar y analizar la especificación reflejada en el documento de especificación de requisitos, obteniendo el *modelo de análisis*.
- Administrar y desarrollar los requisitos a lo largo del proceso de desarrollo.

El proceso de construcción de una *especificación de requisitos* es iterativo. En él, partimos de las especificaciones iniciales imcompletas, poco claras y ambiguas y llegamos a especificaciones finales **completas, claras, documentadas y validadas**.

2.1.1. Concepto de requisito y tipos

Un requisito se puede definir de varias formas:

- Condición o capacidad que debe tener un producto software para resolver la necesidad expresada por un usuario.
- Representación en forma de documento de una capacidad o condición que debe tener un producto software.
- Característica de un productor software que es condición para su aceptación por parte del cliente.
- Propiedad o restricción determinada con precisión que un producto software debe satisfacer.

Tipos de requisito Los requisitos se pueden clasificar según varios factores:

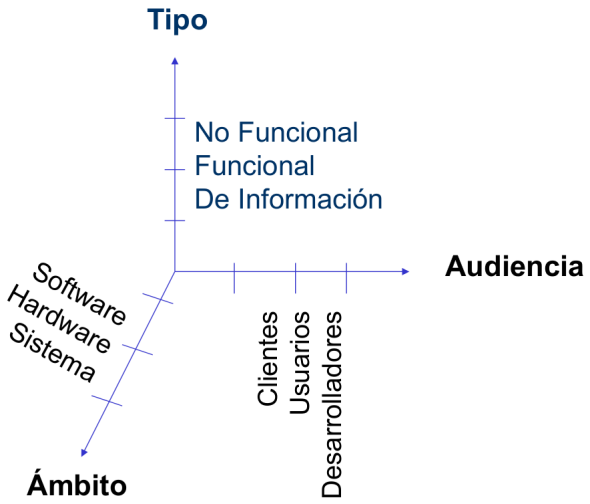


Figura 17: Tipos de requisito.

1. **Funcionales**. Describen la interacción entre el sistema y su entorno, proporcionando servicios que proveerá el sistema o indicando la forma en la que reaccionará ante determinados estímulos.
2. **No funcionales o atributo de calidad**. Describen cualidades o restricciones del sistema que no se relacionan de forma directa con el comportamiento funcional del mismo.
   - Restringen los tipos de soluciones que podemos tomar y suelen restringir el diseño que se realice.
   - No describen funciones, sino propiedades (rendimiento, fiabilidad, seguridad, etc.).
   - Son los que garantizan la calidad del software.
   - Pueden ser requisitos de producto, de organización o externos.
   - Son difíciles de determinar.
3. **De información**. Describen necesidades de almacenamiento de información en el sistema.

Clasificación FURPS+

- Funcionalidad (**F**uncionality): requisito funcional.
- Facilidad de uso (**U**sability): factores humanos, ayuda, documentación.
- Rendimiento (**P**erformance): tiempos de respuesta, productividad, etc.
- Soporte (**S**upportability): adaptabilidad, facilidad de mantenimiento, etc.
- Pseudorrequisitos o restricciones de diseño (+):
  - **Implementación**: limitación de recursos, lenguajes, etc.
  - **Interfaz**: restricciones impuestas para la interacción con sistemas externos.
  - **Operación**: gestión del sistema en su puesta en marcha y a nivel operacional.
  - **Empaquetamiento**: formas de distribución, restricciones de instalación, etc.
  - **Legales**: licencias, derechos de autor, etc.

Ejemplos de requisitos

- El sistema debe validar la tarjeta en menos de tres segundos.
- El sistema debe contar el número de palabras procesadas.
- El sistema se diseñará para un terminal CRT monocromo.
- Los usuarios del sistema serán en su mayoría novatos.
- Deben producirse informes útiles

2.1.2. Propiedades de los requisitos

Para que los requisitos sean de calidad deben satisfacer las siguientes propiedades:

- **Completos**. Todos los aspectos del sistema deben estar representados en el modelo de requisitos.
- **Consistentes**. Los requisitos no deben contradecirse entre sí.
- **No ambiguos**. No se deben poder interpretar los requisitos de varias formas distintas.
- **Correctos**. Deben representar exactamente el sistema que el cliente necesita y que el desarrollador construirá.
- **Realistas**. Los requisitos se deben poder implementar con la tecnología y presupuesto disponible.
- **Verificables**. Se deben poder diseñar pruebas para demostrar que el sistema satisface los requisitos.
- **Trazables**. Cada requisito debe poder rastrearse a través del desarrollo del software hasta su conveniente funcionalidad del sistema.

2.1.3. Tareas de la Ingeniería de Requisitos

1. **Estudio de viabilidad**. ¿Es conveniente desarrollar el sistema?

   - ¿Soluciona el software los problemas existentes?
   - ¿Se puede desarrollar con la tecnología actual?
   - ¿Se puede desarrollar con las restricciones de costo y tiempo?
   - ¿Puede integrarse con otros existentes en la organización?

   Este paso finaliza con la obtención del *informe de viabilidad*.

2. **Obtención de requisitos**. Trabajo con clientes y usuarios para:

   - Estudiar el funcionamiento del sistema.
   - Descubrir las necesidades reales.
   - Consensuar los requisitos entre las distintas partes.

   Es un proceso difícil apoyado por diversas técnicas, como entrevistas, prototipados, casos de uso, etc.

3. **Análisis de requisitos**. Es la actividad más importante. Sus objetivos son:

   - Detectar conflictos entre requisitos.
   - Profundizar en el conocimiento del sistema.
   - Establecer las bases para el diseño.
   - Construir modelos abstractos.

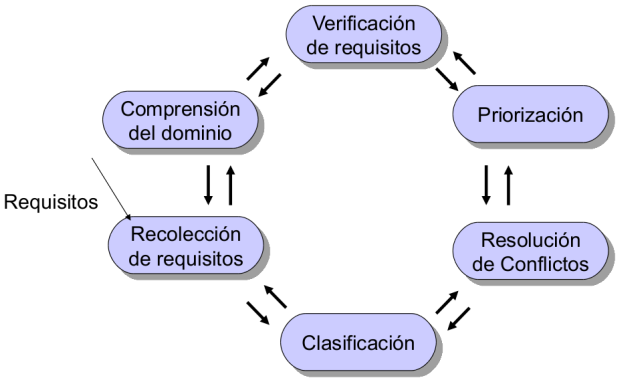


Figura 18: Actividades en el análisis de requisitos.

4. **Especificación de requisitos**. Consiste en representar los requisitos en base al modelo creado en el análisis.

5. **Revisión de requisitos**

   - **Validación**[1]. Consiste en ver que los requisitos reflejan el problema a solucionar.
   - **Verificación**[2]. Consiste en comprobar que la representación sea correcta.

   Es un proceso continuo durante todo el desarrollo.

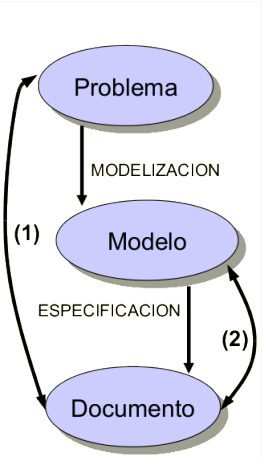


Figura 19: Revisión de requisitos.

En cada fase obtenemos una serie de productos:

- En la **obtención de requisitos**.
  1. Documentos de entrevistas.
  2. Lista estructurada de requisitos.
  3. Diagramas de casos de uso, plantillas de casos de uso y diagramas de actividad.
- En la **especificación de requisitos**.
  - Modelo arquitectónico (subsistemas) → Diagrama de paquetes.
  - Modelo estático (conceptual) → Diagrama de clases.
  - Modelo dinámico (funcional) → Diagrama de secuencia del sistema y contratos.

2.1.4. Roles

En la ingeniería de requisitos podemos distinguir varios roles:

- Stakeholder (personas que tienen relación con el sistema).
- Ingeniero de requisitos.
- Analista de sistemas.
- Arquitecto de software (diseño).
- Documentalista.
- Diseñador de Interfaces de Usuario.
- Gestor de proyecto,
- Revisor.

2.1.5. Problemas de la Ingeniería de Requisitos

Podemos agruparlos en 3 áreas:

- Dificultades para **obtener información**.
- Manejo de la **complejidad del problema**.
- Dificultades para la **integración de los cambios**.

Pueden estar causados por:

- Pobre comunicación
- Uso de técnicas inapropiadas.
- Tendencias a acortar el análisis.
- No considerar alternativas.

2.2. Obtención de Requisitos

2.2.1. Proceso de obtención de requisitos

La obtención de requisitos es la fase inicial de la ingeniería de requisitos. Necesitamos obtener:

- Necesidades y características del sistema.
- Informe del alcance del sistema o producto.
- Lista de participantes.
- Descripción del entorno técnico.
- Lista de los requisitos agrupados por su funcionalidad junto a las correspondientes restricciones que se aplicarán a cada uno.

Las tareas que se deben realizar son las siguientes:

1. Obtener información sobre el dominio del problema y el sistema actual.

   - Conocer el vocabulario propio.
   - Conocer las características principales del dominio.
   - Recopilar información sobre el dominio (consultas con expertos, libros, etc.)
   - Facilitar la comprensión de las necesidades del sistema.
   - Favorecer la confianza del cliente.

   Se ha de entregar la *introducción al sistema* y el *glosario de términos*.

2. Preparar las reuniones de elicitación y negociación.

   - Identificar a los implicados, realizando una descripción general de todos y del perfil de cada uno de ellos.
   - Conocer las necesidades de los clientes y usuarios.
   - Resolver los posibles conflictos

3. Identificar y revisar los objetivos del sistema. Si el sistema es muy complejo, podemos organizarlos mediante una jerarquía. De cada objetivo podemos describir:

   - Su **importancia** (vital, importante o "quedaría bien").
   - Su **urgencia** (inmediatamente, hay presión o puede esperar).
   - Su **estado** durante el desarrollo (en construcción, pendiente de solución, validado, etc.)
   - Su **estabilidad** (alta, media o baja).

4. Identificar y revisar los requisitos de información. De cada requisito podemos describir:

   - Objetivos y otros requisitos asociados.
   - Descripción del requisito.
   - Contenido.
   - Tiempo de vida (medio y máximo).
   - Ocurrencias simultáneas (medio y máximo).
   - Importancia, urgencia, etc.

5. Identificar y revisar los requisitos funcionales. Determinan lo que debe hacer el sistema. De cada requisito podemos describir:

   - Objetivos y requisitos asociados.
   - Secuencia de acciones.
   - Frecuencia.
   - Rendimiento.
   - Importancia, urgencia, etc.

6. Identificar y revisar los requisitos no funcionales. Son las restricciones aplicables a los requisitos funcionales y de información. De cada requisito podemos definir:
    - Descripción.
    - Objetivos y requisitos asociados.
    - Importancia, urgencia, etc.

Tras estos pasos se genera la *lista estructurada de requisitos*.

2.2.2. Técnicas de obtención

- Por **métodos tradicionales**: entrevistas, cuestionarios, análisis de protocolos, etc.
- Por **otros métodos**: técnicas orientadas a puntos de vista, escenarios y casos de uso, etc.

La información la poseen los implicados (*stakeholders*). Un implicado puede ser todo aquel que se beneficia del sistema a construir directa o indirectamente o bien que posea información sobre su funcionamiento o desarrollo, como los responsables del mismo, el cliente, los responsables de la gestión, etc.

2.2.3. Técnicas de entrevista

Tienen el objetivo de obtener información sobre el sistema mediante el dialogo con los expertos en el dominio del problema. Las entrevistas pueden ser de varios tipos: estructuradas o no estructuradas y formales o informales.
Las fases de una entrevista son:

- **Preparación**. Se estudia el dominio del problema, selecciona a los entrevistados y se planifican las entrevistas.
- **Realización**. Consta de:
    - Apertura: presentación e informe sobre los objetivos de la entrevista.
    - Desarrollo.
    - Terminación: recapitulación de la información obtenida.
- **Análisis**. Consiste en reorganizar la información, constatarla con otras fuentes , documentar la entrevista y enviar una copia al entrevistado.

Las principales limitaciones de una entrevista son que lo que los usuarios dicen no es siempre lo que hacen, la timidez y la interpretación de las preguntas. Sin embargo, aporta beneficios como la localización de las áreas en las que profundizar, la involucración de los clientes en el desarrollo o que es una técnica muy conocida y aceptada.

2.2.4. Técnicas de análisis etnográfico

Consiste en observar el contexto del sistema que afecta a los requisitos, es decir, observar la forma en la que las personas trabajan y no como el sistema las hace trabajar.
Hay dos tipos de observaciones:

- **Directa**. El observador está inmerso en el sistema.

- **Indirecta**. Se utilizan entornos de observación.

Se utilizan fundamentalmente para dos tipos de requisitos:

- Los que derivan de la forma en la que trabajan realmente y no de cómo se han definido los procesos.
- Los que derivan de la cooperación y el conocimiento de las actividades de la gente.

No es un enfoque completo, sino que tiene que apoyarse en otras técnicas (entrevistas, prototipado, etc.)

2.3. Modelado de Casos de Uso

2.3.1. Introducción

El modelado de casos de uso es una técnica de la ingeniería de requisitos que permite:

- Delimitar el sistema a estudiar.
- Determinar el contexto de uso del sistema.
- Describir el punto de vista de los usuarios del sistema.

El modelo de casos de uso se utiliza en distintas etapas del desarrollo para:

- Obtener requisitos.
- Analizar y especificar requisitos.
- Como base para el proceso de diseño y su validación.
- Para guiar el diseño de la interfaz de usuario y facilitar la construcción de prototipos.
- Como punto de inicio de las ayudas en línea y el manual de usuario.

Elementos que componen el modelo de casos de uso:

- Actores.
- Casos de uso.
- Relaciones entre:
  - Actores.
  - Actores y casos de uso.
  - Casos de uso.

Para representar y describir estos elementos se utilizan los *diagramas de casos de uso de UML* y las *plantillas estructuradas para actores y casos de uso.*

2.3.2. Diagramas de Casos de Uso

Es un diagrama UML que representa gráficamente todos los elementos que forman parte del modelo de casos de uso junto con la frontera del sistema.

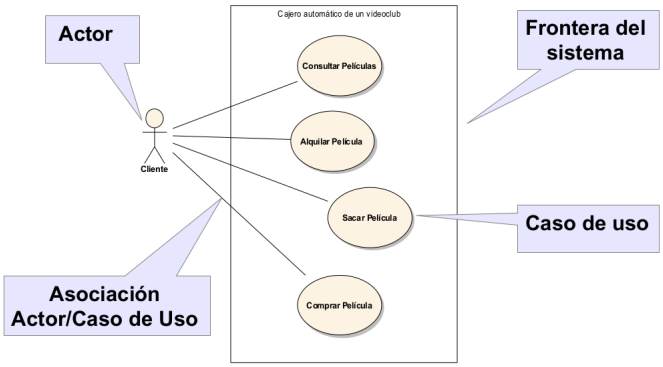
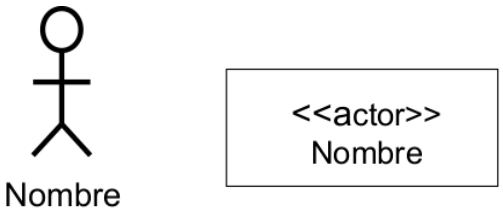


Figura 20: Diagrama de caso de uso

2.3.3. Actor

Definición Un actor es una abstracción de una *entidad externa* al sistema que interactúa directamente con él.

- Los actores especifican roles que adoptan las entidades externas cuando interactúan con el sistema.
- Una entidad puede desempeñar varios roles simultáneamente a lo largo del tiempo.
- Un rol puede ser desempeñado por varias entidades.



(a) (b)

Figura 21: Representaciones de un actor en UML.

Características El nombre del rol debe ser breve y tener sentido desde la perspectiva de negocio. Es frecuente que coincidan con áreas de la empresa (vendedor, gestor de almacén) o distintos niveles de jerarquía (jefe, empleado, aprendiz).

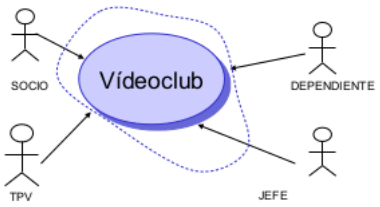


Figura 22: Distintos roles para los actores.

Tipos de actores

- **Principales**. Además de interactuar con el caso de uso, son los que lo *activan*.
- **Secundarios**. Interactúan con el caso de uso, pero no lo activan.

Los actores pueden ser:

- **Personas** con el rol de usuario en el sistema.
- **Dispositivos de E/S** como sensores o medidores, simepre que sean independientes de la acción del usuario.
- **Sistemas informáticos externos** con los que se tiene que comunicar.
- **Temporizador** o reloj cuando se hace algo como respuesta a un evento de tiempo de tipo periódico o en un momento determinado, sin que haya un actor que lo active.

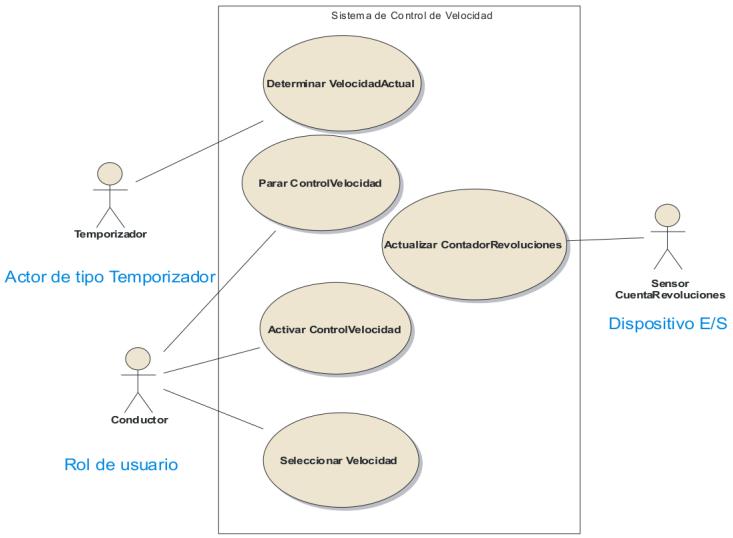


Figura 23: Ejemplos de actores.

Identificación de actores Debemos responder a las siguientes preguntas:

- ¿Quién y qué utiliza el sistema?
- ¿Qué roles desempeñan en la iteración?
- ¿Quién instala el sistema?

- ¿Quién o qué inicio y cierra el sistema?
- ¿Quién mantiene el sistema?
- ¿Qué otros sistemas interactúan con este sistema?
- ¿Quién o qué consigue y proporciona información al sistema?

Relación entre actores

Generalización. Expresa un comportamiento común entre actores, es decir, se relacionan de la misma forma con los mismos casos de uso.

Figura 24: Relación entre actores.

Según esta relación:

- Un empleado modela los aspectos comunes de cualquier tipo de empleado.
- Un jefe hereda los roles y la relaciones de los casos de uso del empleado, además de tener los suyos propios.
- Un actor jefe puede ser usado siempre en lugar de un actor empleado, ya que hereda sus roles y relaciones.

2.3.4. Caso de Uso

Definición Un caso de uso especifica la *secuencia de acciones*, incluidas secuencias variantes y de error, que un sistema o subsistema puede realizar al interactuar con actores externos.

Figura 25: Representación de un caso de uso.

El *nombre* debe ser una frase verbal descriptiva y breve.
Dependiendo de su importancia, los casos de uso pueden ser:

- **Primarios**. Procedimientos comunes más importantes (*procesos de negocio*).

- **Secundarios**. Procesos de error o poco comunes (*procesos internos, diseño*).
- **Opcionales**. Puede que no se implementen.

Características

- Son *iniciados por un actor* que, junto con otros actores, intercambia datos o control con el sistema a a través de él.
- Son descritos desde el *punto de vista de los actores* que interactúan con él.
- *Describen* el proceso de *alcance de un objetivo* de uno o varios actores.
- Tiene que tener una *utilidad real y concreta* para algún actor.
- *Acotan* una *funcionalidad* del sistema.
- *Describen* un fragmento de la *funcionalidad* del sistema de principio a fin → tienen que acabar y dar algún resultado.
- Se documentan con *texto informal*.

Acción del actor		**Acción del sistema**	
1	**Alumno**. Indica que quiere elegir un proyecto determinado.		
2	**Responsable**. Pide al alumno la prioridad con la que se solicita el proyecto.		
		3	Comprueba los proyectos previamente solicitados por el alumno.
		4	Almacena la elección de proyecto del alumno.
		5	Informa de la elección realizada y del éxito de la solicitud.
6	**Responsable**. Informa al alumno de que la solicitud se ha realizado correctamente.		

Figura 26: Ejemplo de caso de uso (elegir proyecto).

Un **caso de uso** sirve para *satisfacer un objetivo* de un actor.
En el análisis de requisitos, un caso de uso se puede descomponer a nivel de *procesos de negocio elementales*.
Un **proceso de negocio elemental** es una tarea o acción realizada por un actor como respuesta a un evento de negocio. Añade un valor cuantificable para el negocio y deja los datos en un estado consistente.

Identificación de casos de uso Debemos responder las siguientes preguntas:

- ¿Qué *objetivos* o *necesidades* tendrá un *actor* específico del sistema?
- El sistema *almacena* y *recupera* información? Si es así, ¿qué actores activan este comportamiento?.
- ¿Qué sucede cuando el sistema *cambia de estado* (por ejemplo, al iniciarlo o detenerlo)? ¿Se le notifica a algún actor?
- ¿Afecta algún *evento externo* al sistema? ¿Cómo se notificarán esos eventos?
- ¿Interactúa el sistema con algún *sistema externo*?
- ¿Generará el sistema algún *informe*?

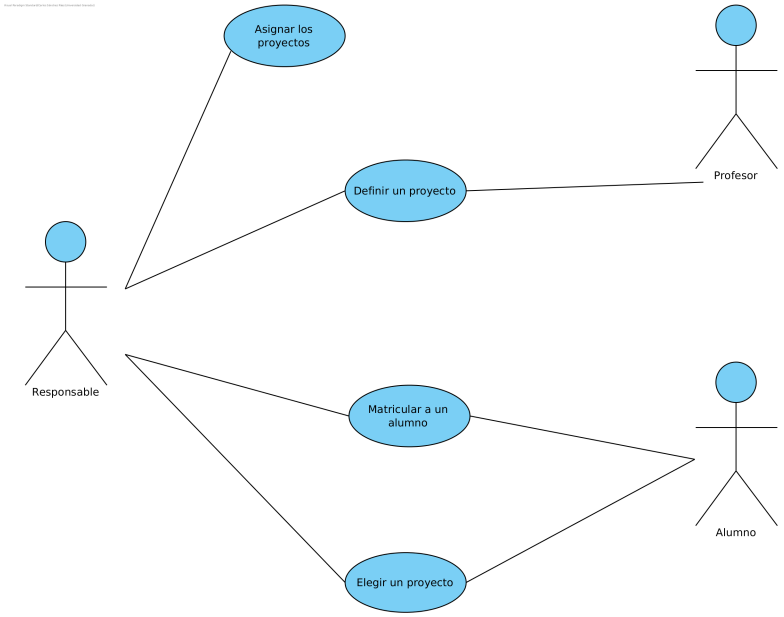


Figura 27: Ejemplo de un caso de uso (gestión de proyectos).

2.3.5. Descripción del actor

Para describir a un actor utilizaremos la siguiente plantilla:

Actor	*Nombre el actor.*				*ID*
Descripción	*Breve descripción del actor.*				
Características	*Características que describen al actor.*				
Relaciones	*Relaciones que posee el actor con otros actores del sistema.*				
Referencias	*Elementos del desarrollo en los que interviene el actor (casos de uso, diagramas, etc.)*				
Autor		**Fecha**		**Versión**	

Atributos		
Nombre	**Descripción**	**Tipo**
...	...	...
Atributos principales del actor.		

Comentarios
Comentarios adicionales sobre el actor.

Figura 28: Plantilla para la descripción de un actor

Actor	Profesor				*ACT1*
Descripción	Profesor que tutoriza algún proyecto de la asignatura TFG				
Características	Puede ser cualquier profesor perteneciente a la ETSIIT.				
Relaciones	-				
Referencias	CU1 (Definir proyecto)				
Autor	LSI	**Fecha**	-	**Versión**	-

Atributos		
Nombre	**Descripción**	**Tipo**
Datos personales	Nombre, DNI, correo, etc.	Lista de cadenas de caracteres.
Departamento	Nombre del departamento al que pertenece	Cadenas de caracteres.
Lista de proyectos	Trabajos que oferta para realizar.	

Comentarios
-

Figura 29: Ejemplo: descripción del actor profesor.

2.3.6. Descripción del Caso de Uso

Contenido La descripción de un caso de uso comprende:

- **Inicio**. Cuándo y qué actor lo produce.
- **Fin**. Cuándo se produce y qué se obtiene.
- **Interacción**. Para qué se usa o qué intenta el caso de uso.
- **Cronología** y origen de las interacciones.
- **Repeticiones de comportamiento**. Qué acciones se repiten.
- **Situaciones opcionales o de error**. Qué situaciones alternativas se presentan en el caso de uso.

Tipos de casos de uso

- Dependiendo del **procesamiento**:
  - **Básico**. Descripción general del procesamiento.
  - **Extendido**. Descripción de la secuencia de acción completa entre actores y sistema.

Figura 30: Diferencia entre caso de uso básico y extendido.

- Dependiendo de su **nivel de abstracción**:
  - **Esencial**. Expresado de forma abstracta, contiene poca tecnología y pocos detalles de diseño.
  - **Real**. Expresado en base al diseño actual. Aparecen relaciones con la *interfaz de usuario*.

Figura 31: Diferencia entre caso de uso real y esencial.

Plantilla básica La plantilla básica para casos de uso es la siguiente:

Caso de uso	*Nombre del caso de uso*			ID	
Actores	*Actores que participan en el caso de uso, indicando si son principales o secundarios.*				
Tipo	*Tipo de caso de uso.*				
Referencias	*Requisitos funcionales*			*Casos de uso*	
Precondición	*Condiciones sobre el estado del sistema que deben cumplirse para que se pueda realizar el caso de uso.*				
Postcondición	*Efectos que tiene la realización del caso de uso sobre el sistema.*				
Autor	-	**Fecha**	-	**Versión**	-

Propósito
Descripción del objetivo que cubre el caso de uso (una línea).

Resumen
Descripción a alto nivel de la secuencia de acciones realizadas en el caso de uso (un párrafo).

Figura 32: Plantilla básica para casos de uso

Caso de uso	Elegir proyecto			CU05	
Actores	Alumno (principal) y responsable (secundario)				
Tipo	Primario y esencial.				
Referencias	RF27 y RF31			CU01	
Precondición	El alumno debe estar matriculado en la asignatura *Proyectos informáticos*.				
Postcondición	El alumno tendrá un proyecto más en su lista de proyectos elegidos.				
Autor	LSI	**Fecha**	-	**Versión**	-

Propósito
Permitir que el alumno pueda elegir un posible proyecto a realizar de entre los ofertados por los profesores.

Resumen
El alumno informa que quiere seleccionar un proyecto, indica la prioridad con la que realiza la selección y se almacena su interés por el proyecto.

Figura 33: Ejemplo: plantilla básica del caso de uso elegir proyecto

Plantilla extendida Para la descripción extendida recurrimos a *escenarios*.

Un **escenario** es una secuencia específica y concreta de acciones e interacciones entre los actores y el sistema objeto de estudio.

Los sistemas pueden ser:

- **Básicos**. Situaciones normales y comunes de actuación.
- **Secundarios**. Se corresponden con situaciones alternativas y de error.

Consiste en añadir lo siguiente a la plantilla básica:

Curso normal			
Actor		**Sistema**	
1	*Acción realizada por un actor*	**2**	*Acción realizada por el sistema.*
..	...	**..**	...
m	*Acción realizada por un actor*	**..**	...
-	...	**n**	*Acción realizada por el sistema.*

Cursos alternos			
n.a	*Alternativa* a *a la acción n del curso normal.*		
Actor		**Sistema**	
1	*Acción del actor*	..	...
..	...	2	*Acción del sistema*
..	...	..	...
n.b	*Alternativa* b *a la acción n del curso normal.*		
..	...	..	...

Otros datos			
Frecuencia esperada	*Número de veces que se realiza el caso de uso por unidad de tiempo.*	**Rendimiento**	*Rendimiento esperado.*
Importancia	*Vital, alta, moderada o baja.*	**Urgencia**	*Urgencia en su desarrollo.*
Estado	*Estado de desarrollo actual.*	**Estabilidad**	*Estabilidad de los requisitos asociados*

Comentarios
Otros aspectos a aclarar.

Figura 34: Plantilla extendida para casos de uso.

Curso normal			
Actor		**Sistema**	
1	El alumno solicita la lista de proyectos ofertados.	**2**	Proporciona la lista.
3	El alumno elige un proyecto determinado.	**4**	Solicita la prioridad con la que el alumno realiza la petición.
4	El alumno proporciona la prioridad deseada.	**6**	Almacena la elección realizada por el alumno e informa de que la operación ha sido realizada con éxito.

Cursos alternos			
6.a	El alumno proporciona una prioridad ya usada para otro proyecto.		
Actor		**Sistema**	
..	...	1	El sistema informa de la situación y finaliza el caso de uso.
6.b	El alumno ha elegido más de 10 proyectos.		
..	...	1	El sistema informa de la situación y finaliza el caso de uso.

Otros datos			
Frecuencia esperada	40 veces al año.	**Rendimiento**	No más de 5 minutos.
Importancia	Alta.	**Urgencia**	Alta.
Estado	Iniciando su desarrollo.	**Estabilidad**	Alta.

Comentarios
El caso de uso se realizará por al menos 40 alumnos.

Figura 35: Ejemplo: plantilla extendida del caso de uso elegir proyecto.

2.3.7. Relaciones de los Casos de Uso

Las *finalidades* de las relaciones son:

- **Organizar** los casos de uso.
- Mejorar la **comprensión**.
- **Reducir la redundancia** del texto.
- **Mejorar la gestión** de los documentos generados.

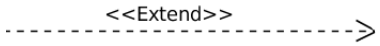
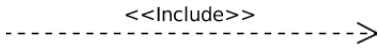
Tipo de Relación	Definición	Notación
Asociación	Relación entre un actor y el caso de uso en el que participa.	
Extensión	Relación entre casos de uso. Representa la inserción de fragmentos de comportamiento adicional sin que el caso de uso base conozca los casos de uso de extensión	
Generalización	Relación entre un caso de uso general y otro específico, que hereday añade características al caso de uso general.	
Inclusión	Relación entre casos de uso. Representa la inserción de comportamiento adicional dentro del caso de uso base. Hace referencia explícita al caso de uso de inclusión.	

Figura 36: Tipos de relaciones entre casos de uso

Relación de inclusión Sus características son las siguientes:

- Es una **relación de dependencia** entre dos casos de uso que permite *incluir el comportamiento* de un caso de uso en el flujo de otro.
- El caso de uso que incluye se denomina *caso de uso base* y el incluido *caso de uso de inclusión*.
- El caso de uso base se realiza hasta que se alcanza el punto donde se encuentra la referencia al caso de uso de inclusión, donde la ejecución pasa al caso de uso de inclusión. Cuando éste termina, el control regresa al caso de uso base.
- El caso de uso de inclusión se utiliza *completamente* por el caso de uso base.
- El caso de uso base *no está completo* sin todos sus casos de uso de inclusión.
- El caso de uso de inclusión puede ser *compartido* por varios casos de uso base.
- El caso de uso de inclusión *no es opcional* y es necesario para que tenga sentido el caso de uso base.

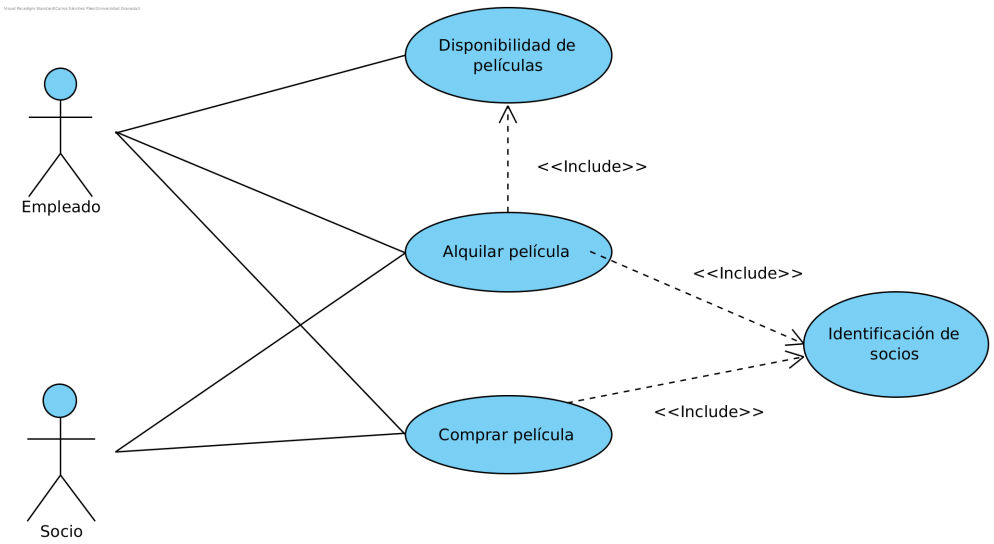


Figura 37: Ejemplo de inclusión en un diagrama.

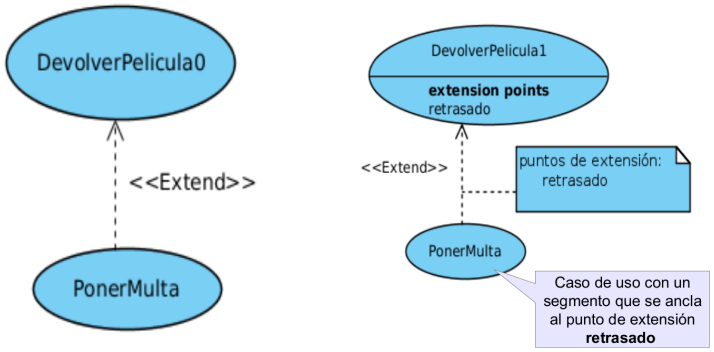
Curso normal			
Actor		**Sistema**	
1	El socio solicita el alquiler de una película.		
2	El socio proporciona sus datos de socio.		
3	El empleado identifica al socio.	**4**	**incluir(CU12. Identificación de socio)**
5	El socio proporciona el título de la película a alquilar.		
5	El empleado identifica la película y pide registro del alquiler.	**7**	**incluir(CU17. Disponibilidad de películas)**.
		8	Guarda los datos del alquiler.
		8	Informa de la cantidad a pagar.
10	El empleado informa al socio de la cantidad a pagar.		
11	Realiza el pago del alquiler.	**12**	Genera el resguardo de alquiler.
10	El empleado entrega el resguardo al socio.		

Figura 38: Ejemplo de inclusión en la descripción de un caso de uso (alquilar película)

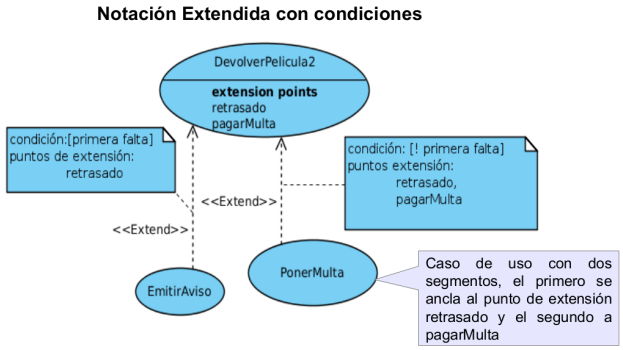
Relación de extensión Sus características son:

- Es una *relación de dependencia* que especifica que el comportamiento del caso de uso base puede ser extendido por otro caso de uso (caso de uso de extensión) bajo determinadas condiciones.
- El caso de uso base declara uno o más *puntos de extensión* que son .ªnclajes.en los que se pueden añadir nuevos comportamientos.
- El caso de uso de extensión define *segmentos de inserción*, que pueden ser insertados en los puntos de enganche cuando se cumpla una determinada condición.

- El *caso de uso base* no sabe nada de los casos de uso de extensión y *está completo sin ellos*. De hecho, los puntos de extensión no están numerados en el flujo de eventos del caso de uso base.
- El caso de uso de extensión no tiene sentido sin el caso de uso base.



(a) Notación básica (b) Notación extendida



(c) Notación extendida con condiciones

Figura 39: Distintas notaciones de la relación de extensión.

Curso normal			
Actor		**Sistema**	
1	El socio solicita devolver de una película.		
2	El socio proporciona sus datos de socio.		
3	El empleado identifica al socio.	**4**	incluir(CU12. Identificación de socio)
5	El socio proporciona el título de la película a devolver.		
	Punto de extensión: retrasado		
6	El empleado incluye la película devuelta.	**7**	Almacena la devolución.
	Punto de extensión: pagarMulta.		
		8	El empleado proporciona justificante de devolución.

Figura 40: Ejemplo de extensión en la descripción de un caso de uso (devolver película)

Caso de uso de extensión	Emitir aviso
Segmento	1
Precondición	Devolución fuera de plazo.

Curso normal de eventos			
Actor		**Sistema**	
1	El empleado incorpora los detalles del aviso.	2	Almacena el aviso.
3	El empleado le indica al socio que tiene un aviso por retraso.		

Figura 41: Ejemplo de descripción de un caso de uso de extensión (emitir aviso)

Para usar correctamente estas relaciones debemos:

- Usar relaciones de *extensión* para comportamientos excepcionales, opcionales o que rara vez suceden.
- Usar relaciones de *inclusión* para comportamientos que se comparten entre dos o más casos de uso, o bien para separar un caso de uso en subunidades.

Relación de generalización

- Es una relación entre un caso de uso general (*padre*) y otros más especializados (*hijos*).
- Los casos de uso hijos:
  - Heredan todas las características del caso de uso general.
  - Pueden añadir nuevas características.
  - Pueden anular o reescribir características del caso de uso general, salvo relaciones, puntos de extensión y precondiciones.

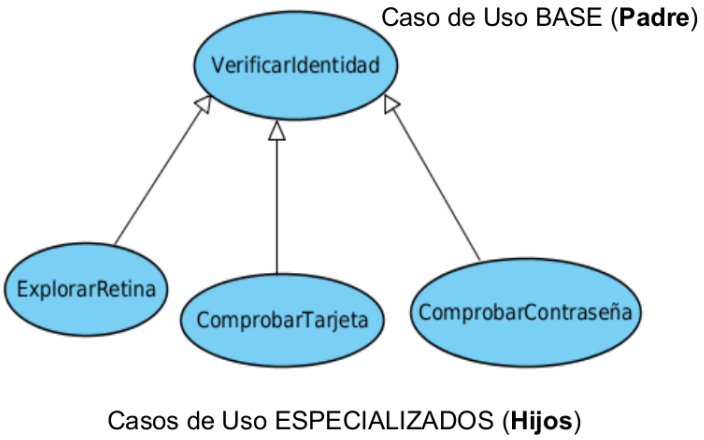


Figura 42: Notación UML de la relación de generalización.

En cuanto a la descripción de los casos de uso hijos no hay diferencia con respecta a cualquier otro caso de uso.

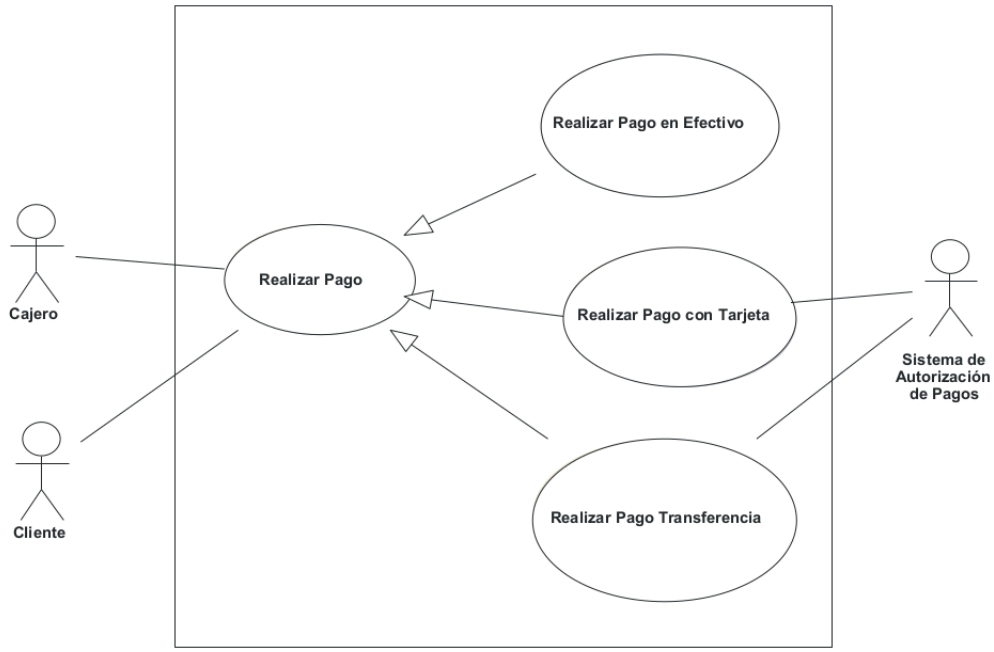


Figura 43: Ejemplo de generalización.

Recomendaciones de uso de las relaciones entre casos de uso

- Usar las relaciones entre casos de uso cuando *simplifiquen el modelo.*
- Un sencillo modelo de casos de uso es preferible a uno con demasiadas relaciones, ya que son más fáciles de entender.
- El uso de muchas relaciones de inclusión hace que tengamos que ver más de un caso de uso para tener una idea completa.
- Las relaciones de extensión son complejas y difícules de entender por la comunidad de usuarios y clientes.
- La *generalización* de casos de uso debería evitarse.

2.3.8. Proceso de construcción del modelo de Casos de Uso

Los pasos a seguir son los siguientes:

1. Identificar actores (*principales* y secundarios).
2. Identificar los principales casos de uso de cada actor, identificando sus objetivos y necesidades, para lo que nos preguntamos:
   a) ¿Cuáles son las *tareas principales* que realiza cada actor?
   b) ¿Qué *información del sistema* debe adquirir, producir o cambiar?
   c) ¿El actor tiene que informar sobre cambios producidos en el *exterior del sistema*?
   d) ¿Qué información desea *adquirir* el actor del sistema?
   e) ¿Desea el actor ser informado de *cambios producidos* en el sistema?

3. Identificar nuevos casos de uso a partir de los existentes, para lo que se deben analizar las siguientes situaciones:

   a) Variaciones significativas de los casos de uso existentes.

   b) Acciones opuestas → casos de uso opuestos a los existentes.

   c) Acciones que deben realizarse antes o después de un caso de uso existente.

4. Elaborar los diagramas de casos de uso y de paquetes en los que se muestren las relaciones lógicas entre diagramas de casos de uso.

5. Confeccionar la descripción básica de cada caso de uso.

6. Definir prioridades y seleccionar casos de uso primarios, teniendo en cuenta:

   - Requisitos imprescindibles.
   - Requisitos importantes.
   - Requisitos deseables.

7. Realizar la descripción extendida de cada caso de uso.

8. Elaborar los diagramas de actividad.

9. Desarrollar prototipos de la interfaz de usuario.

2.3.9. Otros aspectos del modelo de Casos de Uso

Diagrama de paquetes Es un diagrama UML usado para describir la estructura de un sistema mediante agrupaciones lógicas. También muestra las dependencias entre estas agrupaciones. El diagrama de paquetes se utiliza en el modelado de casos de uso para agrupar de forma lógica los distintos diagramas de casos de uso.

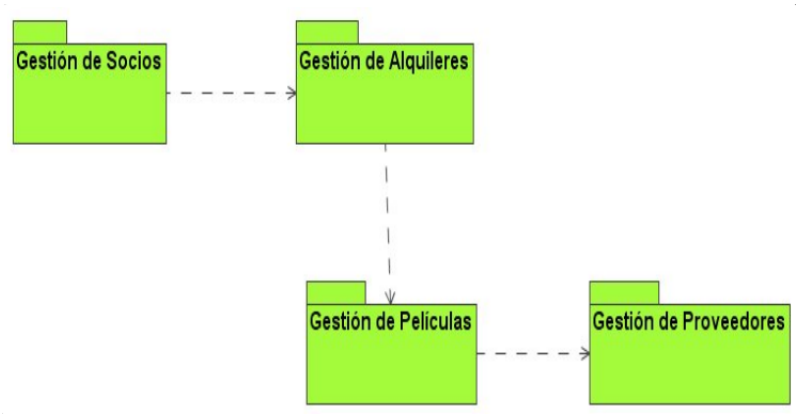


Figura 44: Diagrama de paquetes.

Diagrama de actividad Es un diagrama UML que describe el comportamiento que tienen una serie de tareas o procesos. Representan:

- Los flujos de actividades de los procesos de negocio de una empresa.
- Los flujos de acciones de uno o varios casos de uso de forma gráfica.

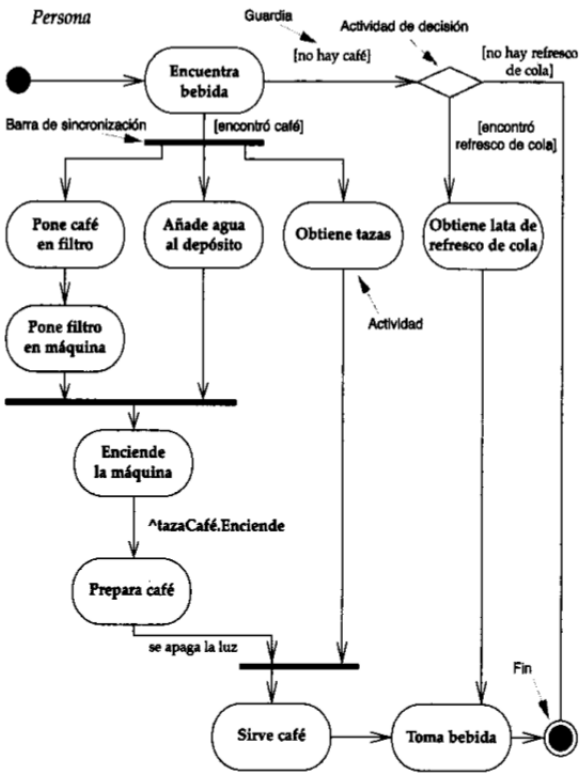


Figura 45: Diagrama de actividad.

2.4. Análisis y especificación de requisitos

2.4.1. Introducción

El proceso de análisis se ha usado tradicionalmente como un sinónimo de *Ingeniería de Requisitos*. Sin embargo, en realidad se trata de una de sus fases, en la que hay que:

- Descubrir los conflictos existentes entre los requisitos.
- Profundizar en el conocimiento del sistema (realizando modelos que sirvan de base para el diseño e implementación y sean más fáciles de entender por desarrolladores).
- Aumentar la formalización del conocimiento existente sobre el sistema para facilitar el mantenimiento.

El objetivo principal del análisis es refinar, estructurar y describir los requisitos para conseguir una comprensión más precisa y fácil de mantener que nos ayude a estructurar el sistema completo mediante modelos de análisis. Es importante rastrear los requisitos de usuario a través de los requisitos del software.

Ejemplo de estudio de soluciones.
Problema. Llevar un control de los productos que se tienen en un almacén y realizar pedidos cuando sea necesario.

Posibles soluciones:

1. Incluir una función en el sistema para obtener un listado de las existencias del almacén de cada producto. El almacenista pedirá los productos de los que haya pocas existencias.
2. Incluir información sobre los mínimos necesarios para cada producto y una función para obtener un listado de los productos bajo mínimo.
3. Incluir información sobre los proveedores de los productos y permitir que el sistema evalue los mínimos cada cierto tiempo, generando listados con los pedidos.
4. Generar los pedidos por FAX de forma automática en base a la información de los proveedores y a los mínimos del almacén.

Modelo CU.	**Modelo de análisis**.
Lenguaje del cliente.	Lenguaje del desarrollador.
Vista externa del sistema estructurado en CU.	Vista interna del sistema estructurado en clases y subsistemas.
Contrato cliente/desarrolladores.	Con vistas a la solución.
Puede contener redundancias e inconsistencias entre requisitos.	No debe contenerlas.
Captura la funcionalidad del sistema	Esboza cómo llevar a cabo la funcionalidad (primera aproximación a la arquitectura).
Se definen CU que serán analizados en mayor profundidad	Define realaciones entre casos de uso.

Figura 46: Diferencias entre modelos.

2.4.2. Análisis orientado a objetos (AOO)

El análisis orientado a objetos examina y representa los requisitos desde la perspectiva de los objetos que nos encontramos en el dominio del problema.

Hay una gran variedad de métodos AOO, aunque todos se centran en la obtención de modelos estáticos (de estructura) y dinámicos (de comportamiento).

El lenguaje más utilizado para representar estos modelos es UML.

El análisis orientado a objetos se usa por los siguientes motivos:

- Los términos usados en los modelos están cercanos a los del mundo real, por lo que facilita y mejora la obtención de requisitos y acerca el espacio del problema al de la solución.
- Se modelan tanto elementos y propiedades estáticas como dinámicas.
- Manejamos conceptos comunes durante el análisis, diseño e implementación del software.

2.4.3. Modelos del AOO

2.4.4. Modelo estático. Diagrama conceptual

- También se le conoce como diagrama de conceptos, diagrama de análisis, diagrama conceptual, modelo conceptual, modelo del dominio, etc.
- En él se representan los principales conceptos del dominio del problema, sus propiedades y las relaciones entre ellos.
- El modelo de casos de uso es la base para obtener la información necesaria para el modelo estático.
- Se representa mediante diagramas de UML.
    - Clases = conceptos del dominio del problema.
    - Asociaciones entre conceptos.
    - Generalizaciones de conceptos.
    - Atributos de los conceptos.

Los pasos que hay que seguir para su obtención son los siguientes:

1. Identificar e incorporar **conceptos**.
2. Identificar e incorporar **asociaciones** entre conceptos.
3. Identificar e incorporar **generalizaciones** de conceptos.
4. Identificar e incorporar **atributos** de conceptos.
5. Estructurar y empaquetar el modelo.

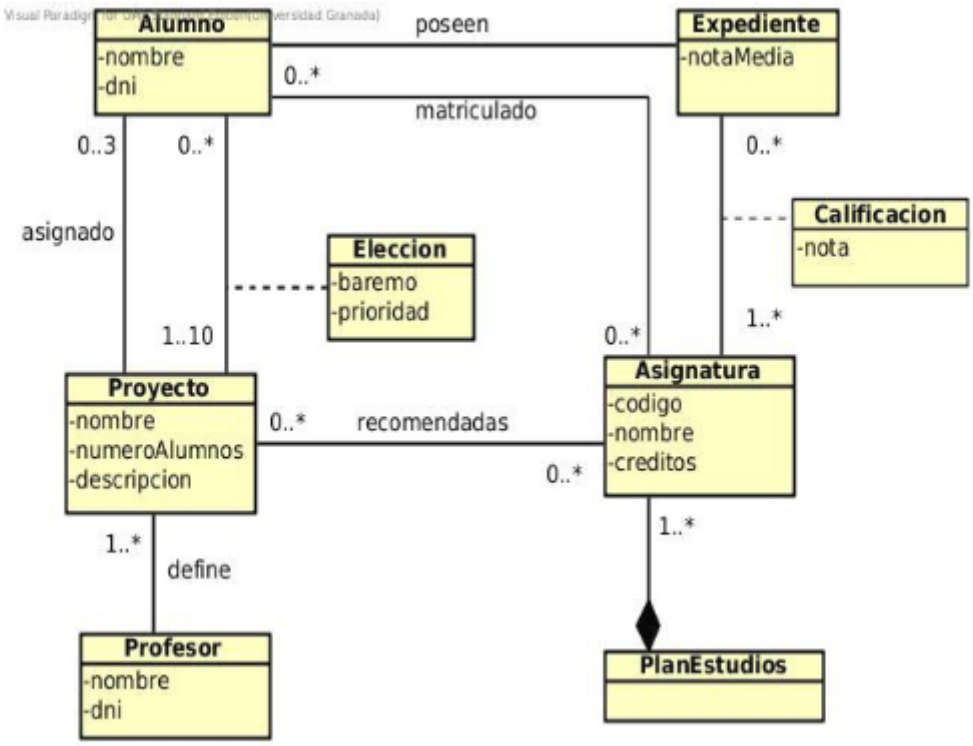


Figura 47: Ejemplo de modelo estático.

2.4.5. Modelo de comportamiento

Es un estudio adicional del dominio del problema en el que añadimos los requisitos funcionales al modelo del análisis.

Diagramas de secuencia del sistema (DSS).

Es un diagrama de secuencia UML en el que sde muestran como los eventos generados por los actores van a provocar la ejecución de una operación por el Sistema, siendo visto como una caja negra.

Para elaborarlos debemos seguir los siguientes pasos:

1. Identificar los **actores** que inician dichas operaciones.
2. Asignar un nombre a todo el sistema.
3. Identificar y nombrar las operaciones principales del sistema de las descripciones de los casos de uso.
4. Ver cuales serían los parámetros de las operaciones.
5. Representarlas en el diagrama de secuencia del sistema (DSS).
6. Incluir las operaciones en la clase que identifica a todo el sistema del diagrama conceptual.

Podemos tener un DSS para cada CU, un solo DSS con todas las operaciones del sistema o bien un DSS por cada diagrama de CU.

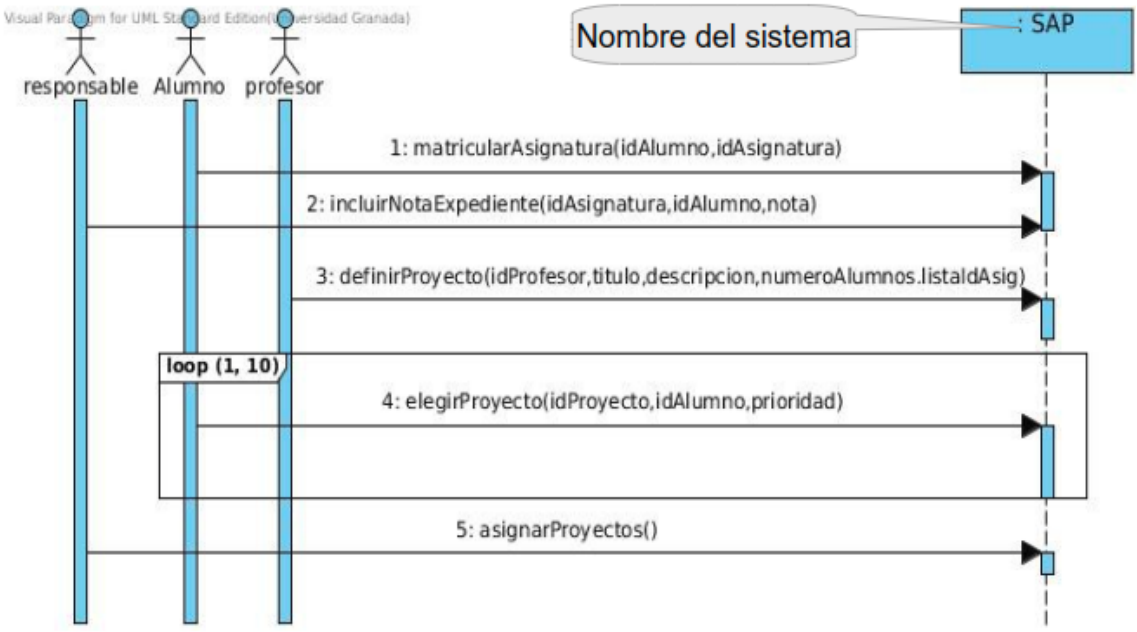


Figura 48: Ejemplo de diagrama de secuencia.

Contratos.

Un contrato es un documento que describe lo que una operación se propone lograr sin decir como se conseguirá. Es decir, define la especificación de una operación sin entrar en su implementación.

La plantilla de un contrato es la siguiente:

Nombre	Nombre de la operación y sus parámetros.
Responsabilidad	Descripción informal de las responsabilidades que debe cumplir la operación.
Tipo	Concepto, clase o interfaz responsable de la operación.
Notas	Notas de diseño, algoritmo, etc.
Excepciones	Casos excepcionales.
Salida	Mensajes o datos que proporciona.
Precondiciones	Suposición acerca del estado del sistema antes de ejecutar la operación.
Poscondiciones	Estado del sistema al terminar la operación. Creación y destrucción de objetos/enlaces, modificación de atributos, etc.

Figura 49: Plantilla de contrato.

Nombre	**definirProyecto**(idProfesor,titulo, descripcion,numeroAlumnas,listIdAsig)
Responsabilidad	Crea un nuevo proyecto inicializando su estado y asignándole el profesor que lo define y las asignaturas recomendadas.
Tipo	SAP
Notas	
Excepciones	▪ Si el profesor identificado por idProfesor no existe. ▪ Si algunas de las asignaturas identificadas por alguno de los elementos de listaIdAsig no existe.
Salida	
Precondiciones	
Poscondiciones	▪ Fue creado un objeto, *pro*, de la clase Proyecto debidamente inicializado. ▪ Fue creado un enlace entre *pro* y el objeto Profesor, identificado por idProfesor. ▪ Para todos los elementos de listaIdAsig: ▪ Fue creado un enlace entre *pro* y el objeto de la clase Asignatura identificado por el correspondiente elemento de listaIdAsig.

Figura 50: Ejemplo de contrato.

3. Tema 3. Diseño e implementación

3.1. Introducción al diseño

3.1.1. Definición y características

EL **diseño** es el proceso de aplicar distintos métodos, herramientas y principios con el propósito de definir un dispositivo, proceso o sistema con los suficientes detalles como para permitir su relización física.

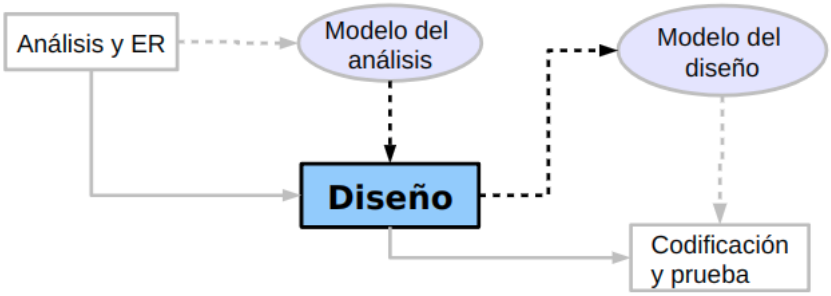


Figura 51: Diseño.

El **diseño del software** es el proceso de aplicar métodos, herramientas y principios de diseño para traducir el modelo de análisis a una representación del software que pueda ser codificada (modelo del diseño).

- El diseño implica una **propuesta de solución** al problema especificado durante el análisis.
- Es una **actividad creativa** apoyada en la experiencia del diseñador.
- Está apoyado por principios, técnicas, herramientas, etc.
- Es una tarea **clave** para la calidad del producto software.
- Es la **base** para el resto de las etapas del desarrollo.
- Debe ser un proceso de **refinamiento**.
- El diseño garantiza que un programa funcione correctamente (*hacer que un programa funcione* $\neq$ *hacer que funcione correctamente).*

3.1.2. Principios de diseño

El diseño se basa en cuatro principios fundamentales: **modularidad**, **abstracción**, **ocultamiento de información** e **independencia modular**.

Modularidad

Consiste en aplicar la estrategia *divide y vencerás*. Un sistema software está formado por piezas (módulos) que encajan perfectamente e interactúan entre sí para llevar a cabo algún objetivo común.

Un **módulo sofware** es una unidad básica de descomposición de un sistema software y representa una entidad o funcionamiento específico. Pueden ser funciones, clases, paquetes, etc.

Las ventajas de la modularidad son:

- Los módulos son más fáciles de documentar que todo el sistema.
- Facilitan los cambios.
- Reducen la complejidad.
- Son más fáciles de implementar.
- Posibilitan el desarrollo en paralelo.
- Permiten la prueba independiente.
- Facilitan el encapsulamiento.

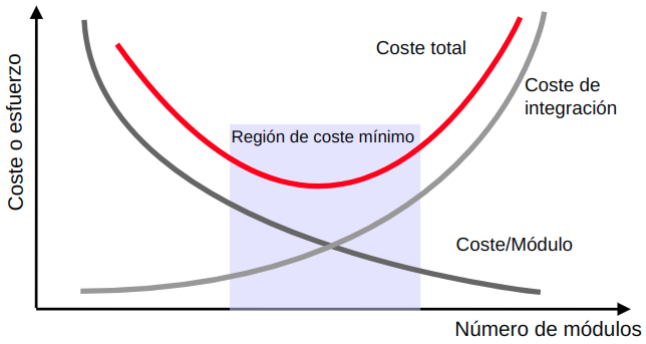


Figura 52: Gráfico adecuado de modularidad.

Abstracción

Es un mecanismo que permite determinar lo que es relavante y lo que no en un nivel de detalle determinado, ayudando a obtener la modularidad necesaria para ese nivel de detalle.

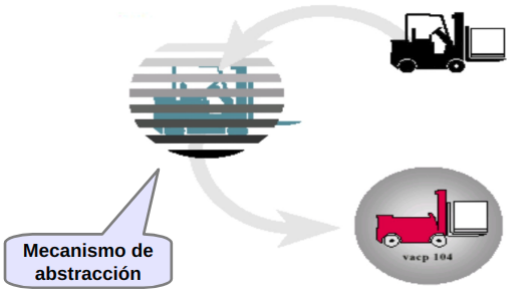


Figura 53: Abstracción.

Existen tres mecanismos fundamentales de abstracción en el diseño: **procedimental**, **de datos** y **de control**.

El proceso de ir incorporando detalles al diseño conforme vayamos bajando el nivel de abstracción se denomina **refinamiento**.

- **Abstracción procedimental**. Consiste en abstraerse sobre el funcionamiento para conseguir una estructura modular basada en procedimientos (métodos).

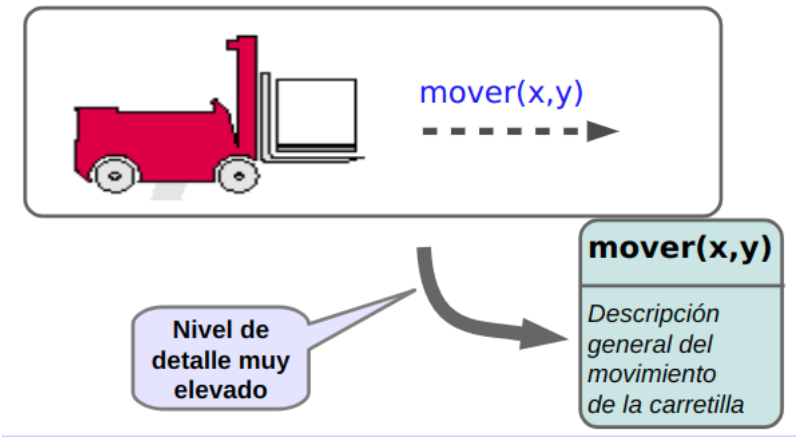


Figura 54: Abstracción procedimental.

- **Abstracción de datos**. Se abstrae tanto el funcionamiento como los atributos que definen el estado de una entidad, obteniendo una estructura modular basada en el estado y el funcionamiento de un objeto o entidad (se añaden variables).

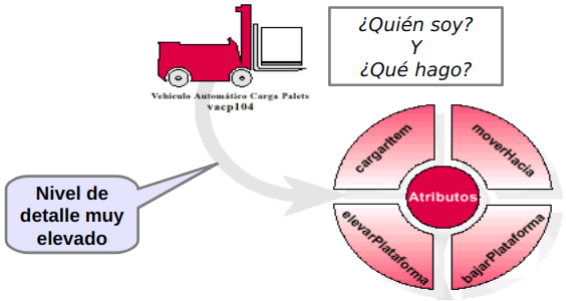


Figura 55: Abstracción de datos.

- **Abstracción de control**. Permikte abstraer sobre el flujo de control de cualquier proceso (definición de los métodos).

Ocultamiento de información

Un módulo debe especificarse y diseñarse de forma que la información (procedimientos y datos) que está dentro del módulo sea inaccesible para otros módulos que no la necesiten.

Las ventajas del ocultamiento de información son:

- Reduce la posibilidad de efectos colaterales.
- Limita el impacto global de las decisiones de diseño locales.
- Enfatiza la comunicación a través de interfaces controladas.
- Disminuye el uso de datos globales.

- Potencia la modularidad.
- Produce software de alta calidad.

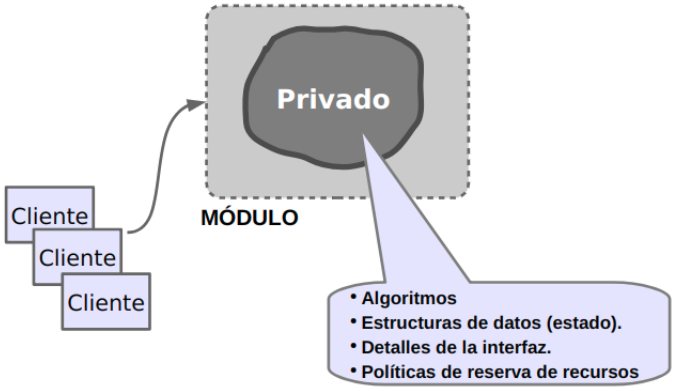


Figura 56: Ocultamiento de información.

Independencia modular

La independencia modular se mide en dos parámetros: **cohesión** y **acoplamiento**:

- **Cohesión**. Grado que tiene un módulo en la realización de *un solo objetivo*. Un módulo debe presentar un alto nivel de cohesión, ya que hace que sea más fácil de entender, reutilizar y mantener.
- **Acoplamiento**. Mide la interdependencia entre módulos dentro de una estructura de software. Un módulo debe presentar un nivel de acoplamiento lo más bajo posible con respecto a los demás módulos.

3.1.3. Herramientas de diseño

Las **herramientas de diseño** son los instrumentos que ayudan a representar los modelos de diseño del software. Algunas de las más usadas son:

- Diagramas UML (de clase, de interacción, de paquetes, etc.).
- Cartas de estructura.
- Tablas de decisión.
- Diagramas de flujo de control.
- Lenguajes de diseño de programas (LDP).

3.1.4. Métodos de diseño

Un **método de diseño** permite obtener diseños de forma sistemática, proporcionando las herramientas, técnicas y pasos a seguir para llevar a cabo del diseño. Todo método de diseño debe poseer:

- **Principios** en los que se basa.
- **Mecanismos de traducción** del modelo de análisis al modelo de diseño.
- **Herramientas** que nos permitan refinar el diseño.
- Criterios para **evaluar** la calidad del diseño.

3.1.5. Modelo de diseño

A nivel general un modelo de diseño está formado por varios **subsistemas** de diseño junto con las interfaces que requieren o proporcionan estos subsistemas. A su vez, cada subsistema puede contener distintos tipos de elementos de modelado de diseño, principalmente *realización de casos de uso-diseño* y *clases de diseño*.

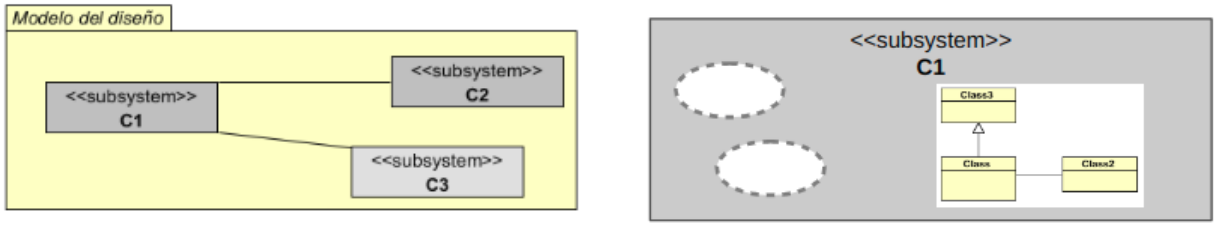


Figura 57: Modelo de diseño.

El modelo de diseño puede considerarse un refinamiento del modelo de análisis en el que todos los artefactos de éste están mejor definidos e incorporan detalles técnicos que permiten su implementación.

Estrategia	**Consecuencias**
Modificar el modelo de análisis.	Se tiene solo un modelo pero se pierde la vista del análisis.
Modificar el modelo de análisis y usar una herramienta que recupere el original	Se tiene un solo modelo, pero la vista recuperada puede no ser satisfactoria.
Congelar el modelo de análisis y hacer una copia para continuar con el diseño.	Se tienen dos modelos que no van al mismo ritmo.
Mantener ambos modelos.	Se tienen dos modelos al mismo tiempo, pero hay sobrecarga de mantenimiento.

Figura 58: Paso de modelo de análisis a modelo de diseño.

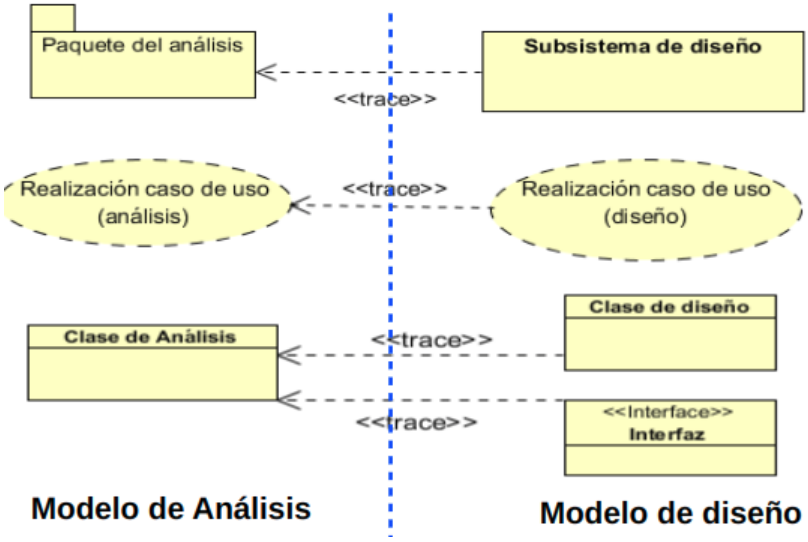


Figura 59: Correspondencia entre modelo de análisis y de diseño.

3.1.6. Tareas del diseño

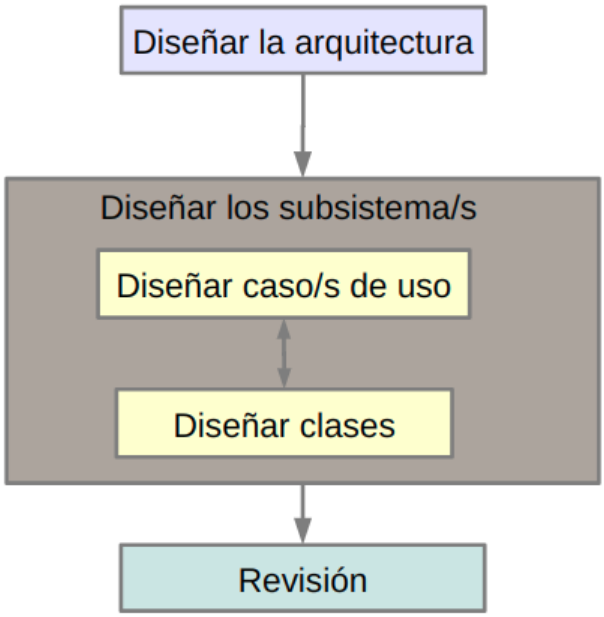


Figura 60: Tareas del diseño.

3.2. Diseño de los casos de uso

3.2.1. Modelo de interacción de objetos

Las herramientas para representar el modelo son los diagramas de interacción UML: el de **secuencia** y el de **comunicación**. Son semánticamente equivalentes.

3.2.2. Patrones de diseño de Craig Larman

Resuelven el problema de la asignación de responsabilidades a objetos.

Una responsabilidad es una obligación que debe tener un objeto en su comportamiento. Todas ellas deben incluirse en alguna operación o método de la clase a la que pertenece el objeto. Las responsabilidades de un objeto pueden ser:

- **Conocer**: datos privados encapsulados en él, objetos relacionados con él, cálculos que puede hacer, etc.
- **Hacer**: iniciar una acción sobre sí mismo o sobre otro objeto, controlar y coordinar actividades entre objetos, etc.

Los **patrones de diseño de Craig Larman** sirven como ayuda para encontrar estas responsabilidades.

Un **patrón de diseño** es la descripción de un problema con su solución en un contexto determinado. Sus pares importantes son:

- Su **nombre**.
- El **problema** que pretende solucionar.
- La **solución** propuesta.
- Las **consecuencias** buenas y malas que puede ocasionar su uso.

Los patrones Craig Larman son varios: **experto en información**, **creador**, **controlador**, **bajo acoplamiento**, **alta cohesión**, polimorfismo, etc.

Nombre	Experto en información
Problema	Complejidad en la búsqueda de información y acoplamientos fuertes entre clases en las búsquedas.
Solución	Asignar responsabilidad a la clase que contiene la información necesaria.
Consecuencias	▪ **Buenas**: mantiene el ocultamiento de información y distribuye el comportamiento. ▪ **Malas**: en ocasiones va en contra de los principios de acoplamiento o cohesión.

Figura 61: Patrón experto en información

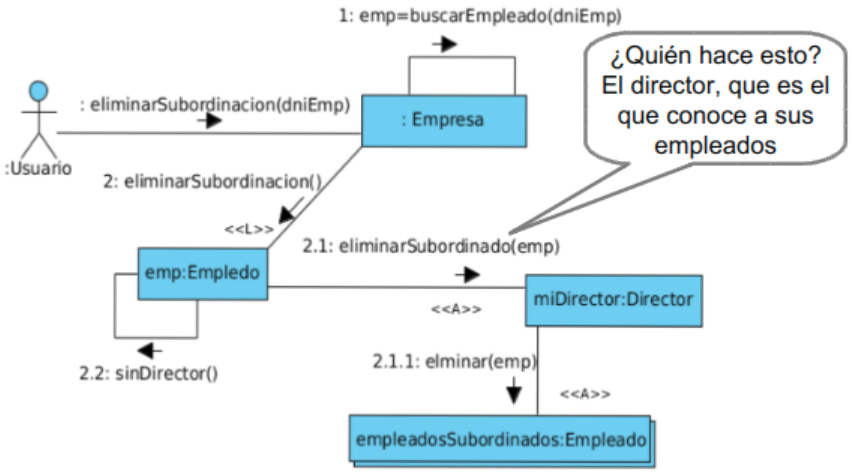


Figura 62: Ejemplo de patrón experto en información.

Nombre	Creador
Problema	Tener acoplamientos, mala encapsulación y reutilización y poca claridad en la construcción de objetos.
Solución	Asignar a la clase B la responsabilidad de crear una instancia de A en caso de que agregue, contenga, registre, utilice o tenga los datos de inicialización de A
Consecuencias	▪ **Buenas**: produce bajo acoplamiento. ▪ **Malas**: no es conveniente usarlo cuando las instancias ya existen.

Figura 63: Patrón creador

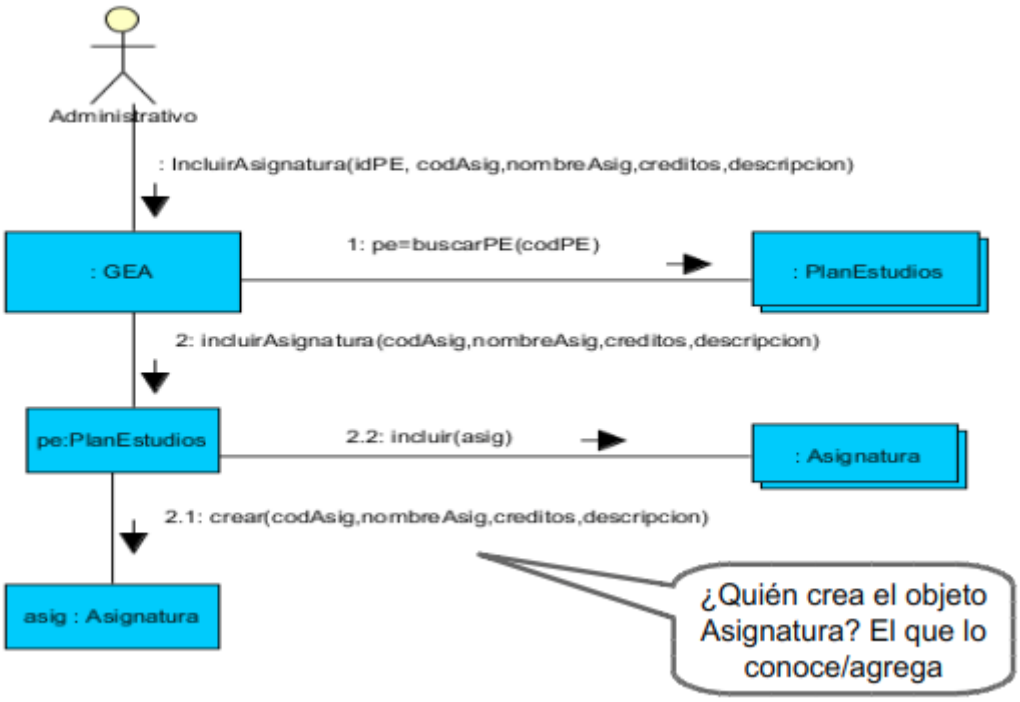


Figura 64: Ejemplo de patrón creador.

Nombre	Bajo acoplamiento
Problema	Elementos que dependen de demasiados elementos. Una modificación conlleva demasiadas modificaciones colaterales.
Solución	Asignar responsabilidades de forma que tengamos elementos que dependan sólo de los elementos necesarios.
Consecuencias	▪ **Buenas**: no afectan los cambios en otros elementos, son fáciles de entender de manera aislada y aumenta la reutilización. ▪ **Malas**: si se lleva a un extremo puede ocasionar diseños pobres. El nivel de acoplamiento debe ser adecuado.

Figura 65: Patrón de bajo acoplamiento

Nombre	Alta cohesión
Problema	Elementos con pocas tareas o con muchas pero no relacionadas.
Solución	Asignar responsabildiades de forma que todas las tareas de un elemento persigan un objetivo común.
Consecuencias	▪ **Buenas**: el diseño es más claro, se simplifica el mantenimiento y aumenta la reutilización. ▪ **Malas**: ninguna, sólo debemos renunciar a este patrón cuando haya un motivo de peso.

Figura 66: Patrón de alta cohesión

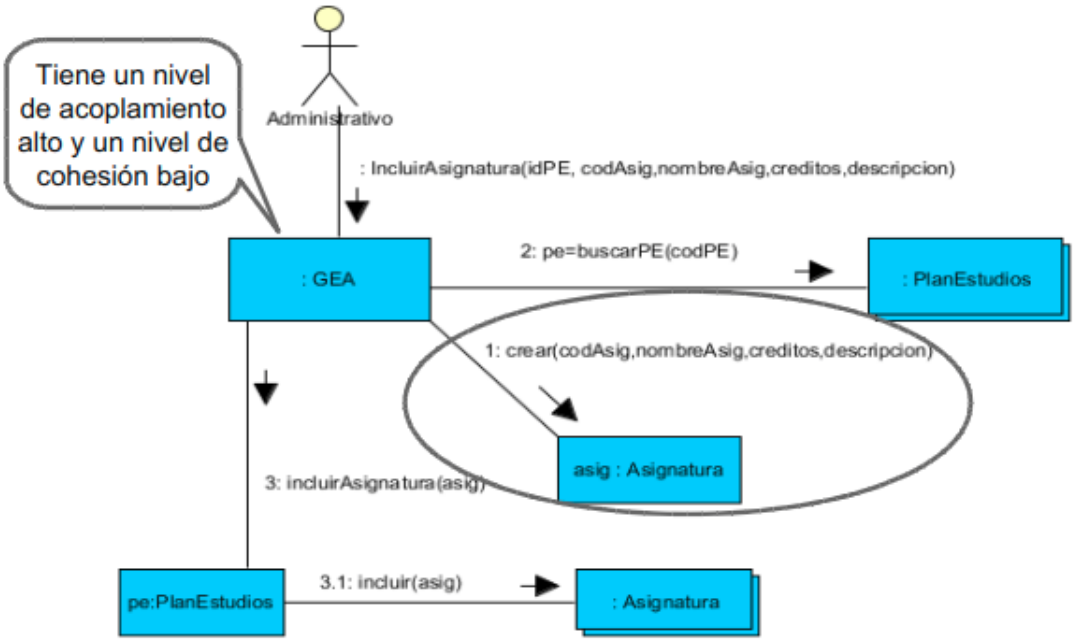


Figura 67: Ejemplo de patrón de alta cohesión.

Nombre	Controlador o fachada
Problema	Comunicación entre los objetos de la capa del dominio de la solución y la capa de la interfaz.
Solución	Asignar responsabilidades de recibir o manejar un mensaje de evento del sistema a una clase que representa al sistema global o bien al caso de uso (un controlador por cada caso de uso).
Consecuencias	▪ **Buenas**: se asegura que la lógica de la aplicación no se maneja en la interfaz. Buena reutilización y bajo nivel de acoplamiento. Permite razonar sobre el estado de los casos de uso. ▪ **Malas**: Controladores saturados.

Figura 68: Patrón controlador o fachada

3.2.3. Elaboración del modelo de interacción de objetos

- Las bases principales para obtener los diagramas de comunicación son los contratos y el modelo conceptual.
- El modelo conceptual sirve como guía para saber qué objetos pueden interactuar en una operación.
- Todo lo especificado en el contrato debe satisfacerse en el diagrama correspondiente.
- Para elaborar cada diagrama podemos ayudarnos de los patrones de diseño de Craig Larman.

Los pasos a seguir son los siguientes:

1. Elaborar los diagramas de interacción teniendo en cuenta el diagrama de conceptos y el contrato asociado.
2. Inicializar el sistema.
3. Establecer las relaciones entre el modelo y la interfaz de usuario.

3.3. Diseño de la estructura de clases

3.3.1. Modelo de estructura de objetos: diagrama de clases del diseño

Un diagrama de clases del diseño describe gráficamente las especificaciones de las clases e interfaces software y las relaciones entre éstas en una aplicación. A diferencia del modelo conceptual, representa la solución a nuestro problema. Puede contener los siguientes elementos:

- Clases con sus atributos y sus operaciones.

- Interfaces con sus operaciones y constantes.
- Relaciones entre clases, interfaces o clases e interfaces.
- Información sobre el tipo de los atributos y parámetros.
- Navegabilidad de las asociaciones.

Se representan mediante un diagrama de clases UML.

3.3.2. Proceso de elaboración del diagrama de clases del diseño

Los pasos a seguir para elaborar el diagrama son los siguientes:

1. Identificar y representar las clases. Todos los objetos de los diagramas comunicación tendrán su correspondiente clase. Tomarán los atributos del modelo conceptual ey de los diagramas de comunicación.
2. Añadir las operaciones. Todos los envíos de mensajes deben tener su operación en la clase correspondiente.
3. Añadir los tipos de atributos y los parámetros.
4. Incluir las asociaciones y la navegabilidad.
5. Incluir las dependencias.

3.4. Diseño de la arquitectura

3.4.1. Conceptos de diseño de la arquitectura

El **diseño de la arquitectura** proporcionará una imagen global de la estructura del sistema software, esbozando los artefactos más importantes que lo componen (principalmente subsistemas y sus interfaces).

Determinar la arquitectura del software consiste en tomar decisiones respecto a:

- Cómo se dividirá el sistema en subsistemas.
- Cómo deberán interactuar dichos subsistemas.
- Cuál será la interfaz de cada subsistema.
- Cuándo será el estilo arquitectónico usado.

La arquitectura es muy importante debido a que:

- Facilita la comprensión de la estructura global del sistema.
- Permite trabajar en los subsistemas de forma independiente.
- Facilita las posibles extensiones del sistema.
- Facilita la reutilización de los distintos subsistemas.

3.4.2. Herramientas para su representación

- **Diagrama de paquetes**. Describe el sistema en torno a agrupaciones lógicas y proporciona una primera estructura.
- **Diagrama de compontentes**. Representa una estructuración concreta del sistema a partir de los sistemas (componentes software) y su correspondiente interfaz (interrelación)
- **Diagrama de despliegue**. Especifica el hardware físico sobre el que se ejecutará el sistema software y cómo cada uno de los subsistemas software se despliega en ese hardware.

3.4.3. Estilos arquitectónicos

Un estilo arquitectónico proporciona un conjunto de subsistemas predefinidos, especificando sus responsabilidades e incluyendo reglas y guías para organizar las relaciones entre ellos. No proporcionan la arquitectura del sistema, sino una guía para obtenerla.

Arquitectura multicapa

Distribuye los subsistemas en capas, de forma que:

- Cada capa presta servicios a la capa superior y actúa como cliente sobre las capas más internas.
- El diseño incluye los protocolos que establecen cómo interactuará cada par de capas.
- Las capas más claras proporcionan servicio de bajo nivel (comunicación en red, acceso a bases de datos, etc.)

Sus principales ventajas son:

- Cada capa puede diseñarse, implementarse y probarse de forma independiente.
- La arquitectura es fácilmente portable: podemos cambiar una capa si mantenemos la interfaz y los cambios en la interfaz sólo afectan a la capa adyacente.

Sus principales ventajas son la dificultad para determinar los servicios de cada capa y la pérdida de rendimiento debida a los múltiples niveles que hay que pasar para completar una petición.

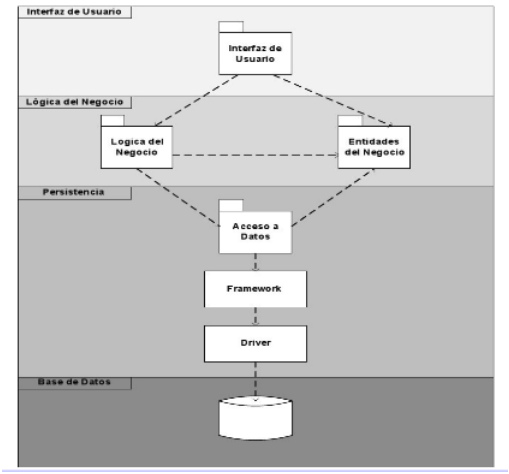


Figura 69: Arquitectura multicapa.

- Las capas pueden diseñarse, construirse y probarse de forma **independiente**.
- Una capa bien diseñada tiene **alta cohesión**.
- Las capas deben estar lo más desacopladas posible.
- Una capa de un nivel no debe tener conocimiento de las capas superiores (**ocultamiento de información**).
- Las capas más inferiores deben prestar servicios de bajo nivel (**bajo acoplamiento**).

Arquitectura cliente-servidor

Están compuestas por:

- **Servidores** independientes que ofrecen servicios a otros subsistemas.
- **Clientes** que pueden solicitar estos servicios.
- **Red** que permite a los clientes realizar peticiones a los servidores.

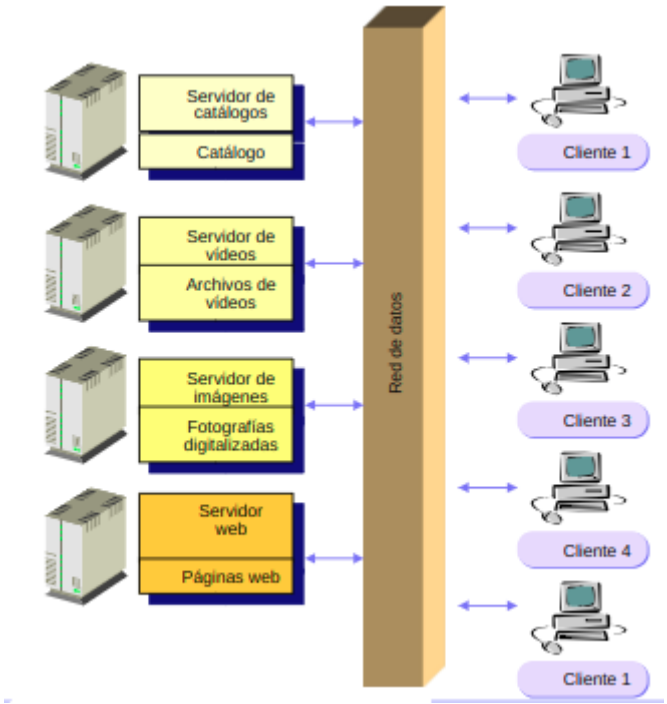


Figura 70: Arquitectura cliente-servidor.

Las capas fundamentales son: de presentación (**clientes**) y procesamiento de aplicación y administración de datos (**servidores**).

- La descomposición en clientes y servidores permite realizar el diseño de ambas partes de forma **independiente**.
- Mejora la **cohesión** de los subsistemas al proporcionar servicios específicos a los clientes.
- Reduce el **acoplamiento** al establecer la red de comunicación.

- Aumenta el nivel de **abstracción** al tener subsistemas o componentes distribuidos en nodos separados.
- Facilita la **reutilización**.
- Los sistemas distribuidos pueden **reconfigurarse fácilmente** añadiendo servidores o clientes extras.
- Pueden escribirse clientes para nuevas plataformas sin modificar los servidores.

Arquitectura de repositorio

Los datos se ubican en una base de datos central a la que acceden todos los subsistemas. Los repositorios pueden ser activos o pasivos.

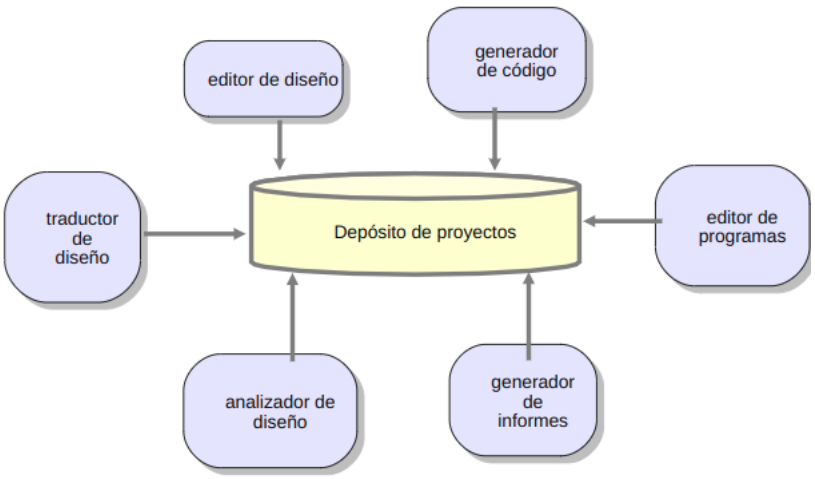


Figura 71: Arquitectura de repositorio.

Sus principales ventajas son:

- Es una forma eficiente de compartir grandes cantidades de datos.
- Los subsistemas productores de datos no necesitan saber de los consumidores.
- Seguridad, control de acceso y recuperación de datos están centralizadas.

Sus inconvenientes son:

- Todos los subsistemas deben estar acordes con el modelo de depósito. Es difícil integrar nuevos subsistemas.
- Es difícil evolucionar hacia otro modelo de datos.
- Fuerza la política de seguridad, control de acceso y recuperación de los subsistemas.
- Se pueden diseñar subsistemas independientes, aunque deben conocer el esquema de los datos del repositorio.
- Mejora la **cohesión** de los subsistemas.
- El **acoplamiento** es alto, haciendo difíciles los cambios.

Arquitectura MVC

Consta de tres componentes, cada uno con un contenido determinado:

- **Modelo**. Responsable del conocimiento del dominio de la aplicación.
- **Vista**. Responsable de mostrar los objetos del dominio de aplicación al usuario.
- **Controlador**. Responde a las acciones del usuario, invocando peticiones al modelo y a la vista.

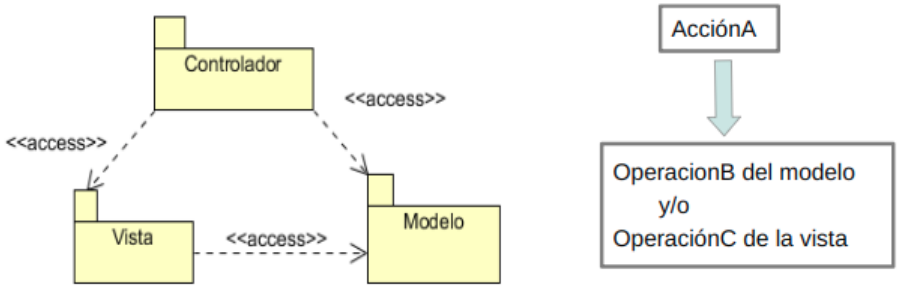


Figura 72: Arquitectura MVC.

- Cada subsistema puede diseñarse de forma **independiente**.
- Se puede aumentar la **cohesión** si la vista y el controlador se encuentran unidos en una capa de interfaz de usuario.
- Se reduce el **acoplamiento**.
- La vista y el controlador hacen uso extensivo de componentes **reutilizables**.
- Es **flexible**: se puede modificar la interfaz de usuario cambiando la vista, el controlador o ambos.
- Se puede probar la aplicación separadamente de la interfaz de usuario.

Arquitectura MDA (*Modern Driven Architecture*)

Permite organizar y gestionar las arquitecturas empresariales. Está basado en herramientas automáticas para construir modelos independientes y transformarlos en implementaciones eficientes.

Los modelos son:

- **CIM** (*Computation Independent Model*). Se encarga del modelado de negocio y requisitos.
- **PIM** (*Platform Independent Model*).
  - Análisis y diseño independiente de la plataforma tecnológica.
  - El analista realiza el PIM que describe al sistema.
  - Un PIM se transforma en uno o varios PSM.
- **PSM** (*Platform Dependent Model*).
  - Análisis y diseño dependiente de la plataforma tecnológica.
  - El arquitecto adapta el modelo para una implementación tecnológica específica.
  - Cada PSM se transforma en código.

3.4.4. Actividades del diseño arquitectónico

Las actividades del diseño arquitectónico son:

1. **Identificar objetivos**.
2. **Determinar y modelar la arquitectura software y los subsistemas**.
3. **Modelar la arquitectura software**.
4. **Refinar la descomposición en subsistemas**.

Identificar objetivos

- Se identifican las cualidades deseables del sistema que determinarán el diseño del mismo.
- Se obtienen a partir de los requisitos no funcionales o del dominio de la aplicación.
- Se selecciona un pequeño conjunto de objetivos que el sistema deberá satisfacer necesariamente.
- Se adquieren compromisos, ya que muchos objetivos de diseño son contrapuestos.

Determinar y modelar la arquitectura

Para **determinarla** hay que:

- Seleccionar el estilo arquitectónico adecuado.
- Identificar subsistemas.
- Añadir subsistemas predefinidos por el estilo arquitectónico elegido.

Para **modelarla** hay que:

- Diseñar la arquitectura en un diagrama de paquetes.
- Elaborar el diagrama de componentes de la vista de arquitectura.
- Realizar el diagrama de despliegue.

Modelar la arquitectura software

Para modelar se utilizan dos tipos de diagramas:

- **Diagrama de paquetes**. Permite modelar una primera estructuración o partición del sistema en base a uno o más paquetes.
- **Diagrama de componentes**. Permite estructurar el sistema en subsistemas en base a componentes que pueden reemplazarse.

Refinar la descomposición en subsistemas

Consiste en refinar los subsistemas hasta satisfacer los objetivos del diseño. Se establecen los siguientes aspectos genéricos:

- Correspondencia hardware-software.
- Administración de datos persistentes.
- Especificación de una política de control de accesos.
- Diseño del flujo de control.
- Diseño de la concurrencia y sincronización.
- Definición de las condiciones del entorno.