Versión 03-07-2020

Apuntes de clase

Análisis y Diseño de Sistemas de Información

(Programador Universitario)

Ingeniería de Software I

(Licenciatura en Informática)

AÑO 2020

Contenidos

Sistema de información

Un sistema de información es un conjunto de datos que interactúan entre sí con un fin común.

En informática, los sistemas de información ayudan a administrar, recolectar, recuperar, procesar, almacenar y distribuir información relevante para los procesos fundamentales y las particularidades de cada organización.

La importancia de un sistema de información radica en la eficiencia en la correlación de una gran cantidad de datos ingresados a través de procesos diseñados para cada área con el objetivo de producir información válida para la posterior toma de decisiones.

Un sistema de información se destaca por su diseño, facilidad de uso, flexibilidad, mantenimiento automático de los registros, apoyo en toma de decisiones críticas.

Todos estos elementos interactúan para procesar los datos (incluidos los procesos manuales y automáticos) y dan lugar a información más elaborada, que se distribuye de la manera más adecuada posible en una determinada organización, en función de sus objetivos.

Habitualmente el término "sistema de información" se usa de manera errónea como sinónimo de “sistema de información informático” ó “sistema informático”, en parte, porque en la actualidad, en la mayoría de los casos los recursos materiales de un sistema de información están constituidos casi en su totalidad por sistemas informáticos. Estrictamente hablando, un sistema de información no tiene por qué disponer de dichos recursos (aunque en la práctica esto no suela ocurrir). Se podría decir entonces que los “sistemas de información informáticos” son una subclase o un subconjunto de los “sistemas de información” en general.

Análisis y diseño de sistemas

Dentro de las organizaciones, el análisis y diseño de sistemas se refiere al **proceso de examinar la situación de una empresa con el propósito de mejorarla con métodos y procedimientos adecuados**.

En términos generales, el desarrollo de sistemas está formado por dos grandes componentes:

a) Análisis de Sistemas: Proceso de clasificación e interpretación de hechos, diagnóstico de problemas y empleo de la información para recomendar mejoras al sistema.

b) Diseño de Sistemas: Es el proceso de planificar, reemplazar o complementar un sistema organizacional existente.

> Solamente después de haber reunido los hechos, el analista se encuentra en posición de determinar cómo y dónde un sistema de información basado en computadora será benéfico para todos los usuarios del mismo. Esta acumulación de información, llamada **estudio del sistema**, es la que precede a todas las actividades del análisis del sistema.

En otras palabras, el análisis especifica **qué** es lo que el sistema deba hacer y el diseño establece **cómo** alcanzar este objetivo.

Objetivos

Etapa 1°. **- Análisis y definición de necesidades**: El objetivo es tomar una visión general del sistema e identificar sus unidades. Consultando con el usuario se establecen los servicios que el sistema debe brindar, las restricciones y objetivos del mismo.

Una vez acordados, deben definirse de una manera comprensible, tanto para los usuarios como para el personal que desarrollará el sistema.

Etapa 2°. **- Diseño del sistema y del software**: Las necesidades de un sistema se clasifican en necesidades de hardware y necesidades de software. El proceso de definirlas se llama **diseño de sistemas**. El diseño de software es el proceso de representar las funciones de

cada sistema de software a fin de poderlo transformar con facilidad en uno o más programas de computación.

¿Qué es un sistema?

En un sentido amplio,

> **Un sistema es un conjunto de componentes que interactúan entre sí para lograr un objetivo común**.

Así, una organización es un sistema en el que sus componentes, por ej. Departamentos de Mercadotecnia, Manufactura, Venta, Contabilidad, Personal, etc., trabajan para crear utilidades que beneficien tanto a los empleados como a los accionistas de la empresa.

Pero todo sistema organizacional depende, en mayor o menor medida, de una entidad abstracta llamada **sistema de información,** que es el medio por el cual los datos fluyen de una persona o departamento hacia otro.

Para lograr sus objetivos, los sistemas interactúan con su **medio ambiente**, formado por todos los objetos que se encuentran fuera de las fronteras del sistema.

Los sistemas se pueden **clasificar como**:

Sistemas Abiertos: Sistemas que reciben entradas y producen salidas, es decir, sistemas que interactúan con el ambiente.

Sistemas Cerrados: Son sistemas que no interactúan con el ambiente. En la actualidad solamente son conceptuales ya que todos los sistemas existentes son abiertos.

Un sistema se encuentra **bajo control** cuando opera dentro de los niveles de desempeño tolerables. Estos niveles aceptables de desempeño se denominan **estándares** y sirven para comparar el nivel de desempeño actual del sistema. Las diferencias se anotan y se reportan en un proceso llamado **retroalimentación**.

Definiciones en general

Sistema de Programación: Se compone de un **conjunto de programas autónomos**, que pueden estar dedicados, o no, a una sola aplicación. (Ejemplos:

	Sistemas Operativos, Sistemas de Mandato y Control, Sistemas de Automatización de oficinas, etc.)
Subsistema:	Es un sistema de programación que **forma parte de otro sistema mayor**, pero que siempre está dedicado a una sola aplicación. (Ejemplo: En un sistema de automatización para oficina pueden existir el subsistema de correo electrónico, el subsistema de contabilidad, etc.)
Programa:	Es la **especificación de la solución a un problema**, que puede ejecutar una computadora.
Proceso:	Es un **programa en ejecución**. En la misma computadora se pueden ejecutar simultáneamente varios procesos.
Objeto de programa:	Cualquier entidad que, en un programa, pueda **recibir un nombre** (variables, constantes, módulos, procedimientos, funciones).
Módulo:	Es una **colección de objetos de programa identificada con un nombre**. Se hace referencia a los objetos de un módulo, especificando el nombre del módulo y el nombre del objeto.
Procedimiento:	**Es un objeto ejecutable de un programa**. Los objetos que se declaran en un procedimiento comienzan a existir cuando se activa el procedimiento y dejan de existir cuando termina su ejecución.
Función:	**Es un tipo particular de procedimiento** que acepta al menos un valor de entrada y devuelve un único valor de salida. Es una abstracción sobre una expresión y, por lo tanto, debe ser invocada desde una expresión.
Unidad de Programa:	Ó **componente de software**. Es un módulo, un procedimiento o una función.

Ciclo de Vida del Desarrollo de Software

El ciclo de vida de desarrollo de software *(Software Development Life Cycle (SDLC))*, también denominado ciclo de vida de desarrollo de aplicaciones, es un **proceso** para **planificar, crear, probar, implementar y mantener un sistema de software**.

El marco de trabajo del ciclo de vida del desarrollo de software, proporciona una secuencia de actividades que desarrolladores y usuarios de software deben seguir. Consiste en un conjunto de pasos o fases en las que cada fase del SDLC utiliza los resultados de la anterior.

Al igual que todo lo que se fabrica en una línea de ensamblaje, el **proceso SDLC** tiene como objetivo producir sistemas de alta calidad que cumplan o superen las expectativas del cliente, en función de los requisitos del cliente, mediante la entrega de sistemas que se mueven a través de cada fase claramente definida, dentro de los plazos programados y las estimaciones de costos.

Los grandes sistemas de software requieren un tiempo considerable para su desarrollo, y permanecen en uso un tiempo aún mayor.

SDLC es un **proceso sistemático**[1] para crear software que mejora la calidad y la corrección del software creado. El desarrollo del sistema debe completarse en el marco de tiempo y costo predefinidos. SDLC consiste en un plan detallado que explica cómo planificar, construir y mantener software específico. Cada fase del ciclo de vida de SDLC tiene su propio proceso y entregables que alimentan la siguiente fase.

El SDLC se adhiere a fases importantes que son esenciales para los desarrolladores, como la planificación, el análisis, el diseño y la implementación. Al igual que todo lo que se fabrica en una línea de ensamblaje, un SDLC tiene como objetivo producir sistemas de alta calidad que cumplan o superen las expectativas del cliente, en función de los requisitos del cliente, mediante la entrega de sistemas que se mueven a través de cada fase claramente definida, dentro de los plazos programados y las estimaciones de costos.

El SDLC brinda a las organizaciones un **enfoque metódico**, paso a paso, para desarrollar software exitoso. Desde reunir los requisitos iniciales para un nuevo producto, hasta mantener un producto maduro en el mercado.

Con el tiempo, se han adoptado variaciones del marco de trabajo para el desarrollo de productos de tecnología de software.

[1] proceso sistemático: Es un proceso que se encuentra regido por una serie de pasos, los cuales siguen un orden determinado de ejecución, organizados de forma lógica, para el logro de un determinado fin.

El proceso SDLC se divide en las siguientes 7 fases o etapas:

1. Recopilación y análisis de requisitos
2. Estudio de Factibilidad
3. Diseño
4. Codificación
5. Pruebas
6. Instalación e Implementación
7. Mantenimiento

La fase de mantenimiento puede originar cambios en las etapas anteriores, como por ejemplo, nuevas necesidades del cliente, nuevos diseños, pruebas adicionales al sistema, por lo que es importante tener en claro que dicha fase está relacionada a todas las anteriores.

Fase 1: Recopilación y análisis de requisitos

En esta etapa el analista trata de tomar una **visión general del sistema** e identificar sus unidades, **y de los servicios que el sistema debe brindar**. Las restricciones y objetivos del sistema se establecen consultando con los usuarios. Una vez acordados, deben definirse de una manera comprensible, tanto para los usuarios como para el personal que desarrollará el sistema.

Durante esta etapa el Ingeniero de Software actúa como **asesor técnico del usuario**, en relación a un proyecto particular.

El **requisito,** es sin dudas, el objeto de estudio en la primera etapa del proceso SDLC. Lo llevan a cabo los miembros senior del equipo con aportes de todas las partes interesadas y expertos en dominios de la industria. La **planificación de los requisitos** da garantía de calidad y el reconocimiento de los riesgos involucrados también se realiza en esta etapa.

Esta etapa permite tener una visión más clara del alcance de todo el proyecto y los problemas, oportunidades y directivas anticipadas que desencadenaron el proyecto.

La **etapa de recopilación de requisitos** necesita equipos para obtener requisitos detallados y precisos. Esto ayuda a las empresas a finalizar el cronograma necesario para finalizar el trabajo de ese sistema.

La **recopilación y análisis de los requisitos del sistema** implica reconocer y comprender las facetas importantes del sistema que está bajo estudio. Esto, de alguna manera, implica responder a las siguientes preguntas:

a) ¿Qué es lo que hace?

b) ¿Cómo lo hace?

c) ¿Con qué frecuencia se presenta?

d) ¿Qué tan grande es el volumen de transacciones o de decisiones?

e) ¿Cuál es el grado de eficiencia con que se realizan las tareas?

f) ¿Existe algún problema?

g) ¿Si existe, ¿qué tan serio es?

h) ¿Si existe, ¿cuál es la causa que lo origina?

Estas preguntas pueden ser respondidas por el analista luego de conversar con distintas personas de la empresa para obtener detalles relacionados con los procesos de la empresa, opiniones, soluciones propuestas e ideas para cambiar el proceso.

La recopilación de requisitos es el estudio de un sistema para conocer cómo trabaja y donde es necesario efectuar mejoras.

Un **requisito es una característica** que debe incluirse en un nuevo sistema. La inclusión se hace de determinada forma para capturar o procesar datos, producir información, controlar una actividad de la empresa o brindar un soporte a la gerencia. Así, el analista debe antes que nada comprender la situación, entender y clasificar los requisitos del sistema en aquellos que son comunes a casi todas las situaciones y los que son particulares del sistema.

Para poder **determinar requisitos de un sistema** se llevan a cabo tres grandes **actividades**:

Anticipación de requisitos: Prever las características del sistema en base a la experiencia previa. El analista debe investigar áreas y aspectos, aunque esto pueda introducir un sesgo propio de la experiencia que tenga el analista.

Investigación de requisitos: Estudio y documentación del sistema actual usando técnicas para encontrar hechos, análisis del flujo de datos y análisis de decisión.

Especificación de requisitos: Los datos obtenidos durante la recopilación se analizan para determinar las especificaciones de requisitos, es decir, la descripción de las características del nuevo sistema.

Esta última actividad se compone de **tres partes relacionadas entre sí**:

- **Análisis de Datos basados en hechos reales**: Se examinan los datos recopilados durante el estudio, incluidos en la documentación de flujo de datos y análisis de decisiones, para examinar el grado de desempeño del sistema y si cumplen con las demandas de la organización.

- **Identificación de requisitos esenciales:** Características que deben incluirse en el nuevo sistema y que van desde detalles de operación hasta criterios de desempeño.

- **Selección de estrategias para satisfacer los requisitos:** Métodos que serán usados para alcanzar los requisitos establecidos y seleccionados.

Los analistas usan diferentes técnicas, **técnicas para encontrar hechos**, para recopilar información relacionada con los requisitos:

Entrevista: Con la entrevista el analista reúne información proveniente de personas o grupos de personas. Los entrevistados serán los usuarios del sistema existente o los potenciales usuarios del sistema propuesto. Ambos, el analista y los entrevistados, conversan, se hacen preguntas y respuestas. El éxito de la entrevista depende de la habilidad del entrevistador y de su preparación para la entrevista.

La entrevista de Análisis: Es muy importante que la tarea de entrevista sea eficaz y eficiente ya que el 80% de la tarea del analista se basa sobre datos que, oralmente, le brindan sus entrevistados.

El siguiente objetivo es válido para todas las instancias de la entrevista:

El entrevistador desea obtener información del entrevistado

La entrevista se puede dividir en **tres momentos**:

- **el antes**: El entrevistador debe tener una preparación previa: cuáles son las estructuras formales del sector y su relación con otras áreas, flujos de datos (en grueso), el alcance de los cambios propuestos y la historia de las relaciones del sector con el área de sistemas.
- **el durante**: La actitud del analista debe ser receptiva, debe mostrar su interés por el tema, estar alerta a mensajes incompletos o contradictorios y motivar al usuario para que le transmita la información que necesita.
- **el después**: Se propone hacerle llegar al cliente un informe de las notas que él tomó durante la entrevista para solicitarle que las revise y corrija.

Cuestionarios: Permiten al analista reunir información relacionada proveniente de un grupo grande de personas. Se asegura el anonimato de los encuestados, pero también se seleccionan los encuestados en base a asegurarse que sus conocimientos y experiencia los califiquen para dar respuestas a las preguntas.

Revisión de los registros: El analista examina la información asentada en registros y reportes relacionada con el sistema y los usuarios para conocer la organización y sus operaciones.

Observación: Permite al analista ganar información sobre la forma en que se llevan a cabo las actividades, información que no se puede obtener por medio de otras técnicas.

Fundamentos del análisis de requisitos

Siempre ocurre que el cliente trata de formular un concepto algo nebuloso de la función y performance que debe desarrollar el sistema. Es aquí que el ingeniero de

software actúa como un **interrogador**, un **consultor** y **una persona que resuelve problemas**.

• Reconocimiento de Problemas:

Identifica los aspectos de cada problema y sus características particulares.

• Evaluación de problemas y síntesis de la solución:

Se concentra en los datos que el sistema produce, cuáles son las funciones que debe tener, qué interfaces están definidas, qué restricciones se deben aplicar.

• Modelamiento:

Se crea un modelo gráfico y textual del sistema a construir. Este modelo será usado como base de diseño y como punto focal de futuras revisiones. Se pueden usar lenguajes formales de especificación formal de los requisitos.

• Especificación:

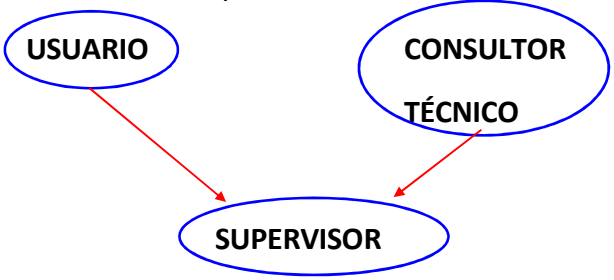
Aquí se produce una especificación de los requisitos que documenta los pasos en la evaluación del problema, una síntesis de la solución y un modelo. Además, incluye una predicción de costos, beneficios, características del sistema y un cronograma de desarrollo.

• Revisión:

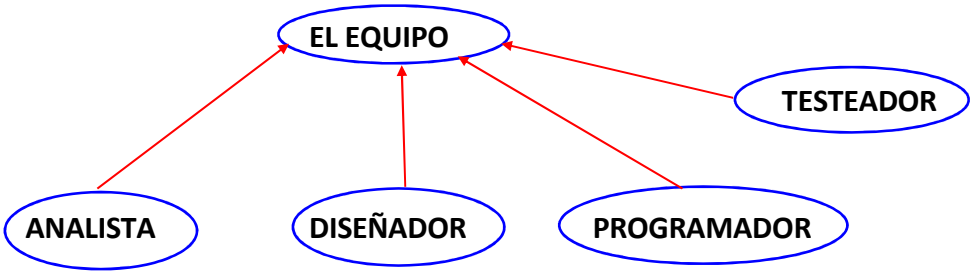
El documento anterior es discutido con el cliente.

Los roles

El Supervisor del equipo de desarrollo tiene asociada una doble tarea: funciona como contacto con el usuario y además como consultor técnico.



La tarea del equipo de desarrollo está sujeta a tres roles que se asignan de acuerdo a las capacidades personales de cada uno de los integrantes:



a) El Analista de Información:

Cuando la tarea asignada al analista de sistema es solamente el análisis del sistema, su tarea es la de conducir estudios del sistema para detectar hechos relevantes relacionados con la actividad de la empresa. Su tarea más relevante es reunir información y determinar requisitos, pero no diseñar sistemas.

b) El analista diseñador de sistemas o aplicaciones:

Además de llevar a cabo el estudio completo del sistema, este rol tiene la responsabilidad adicional de diseñar el nuevo sistema. Los diseñadores de una empresa de software, trabajan en menos cantidad de proyectos pero invierten más tiempo en cada uno de ellos.

c) El analista programador:

Conduce la investigación de sistemas, desarrolla las especificaciones de diseño y escribe y software necesario para implementar el sistema.

Dependiendo del tamaño de la empresa de software, el papel que cumple el analista de sistemas puede ser muy diferente. En cualquier caso, el analista de sistemas más valioso y calificado es el que sabe cómo programar.

Fase 2: Estudio de Factibilidad

Una vez que se completa la fase de análisis de requisitos, el siguiente paso es definir y documentar las necesidades de software. Este proceso se realizó con la ayuda del Documento de Requisitos de Software o Especificación de Requisitos de Software - Software Requirements Specification (SRS). Incluye todo lo que debe diseñarse y desarrollarse durante el ciclo de vida del proyecto.

El estudio de la factibilidad de un proyecto implica estudiar la:

Factibilidad Técnica: Si el proyecto puede realizarse con la tecnología de software existente. ¿El problema es computable? ¿Los algoritmos conocidos para resolverlo son eficientes? ¿Las condiciones de funcionamiento imponen hardware que todavía no se conoce?

Factibilidad Económica: ¿Los beneficios esperados cubren los costos de construir el sistema? ¿Es aceptable el riesgo de no terminar el proyecto? ¿Podemos completar el proyecto dentro del presupuesto o no?

Factibilidad Operativa: ¿A qué se expone la empresa al pasar al nuevo sistema? ¿Será usado? ¿Habrá resistencia al cambio por parte de los usuarios, del personal, de los clientes y del sistema global pre-existente? ¿Podemos crear operaciones que el cliente espera?

Factibilidad Legal: ¿Es legalmente posible realizar lo que el cliente me pide? Estudio de la legislación correspondiente.

Factibilidad Temporal: ¿Puede el proyecto ser completado en el tiempo dado?

Fase 3: Diseño

En esta etapa se producen los detalles que establecen la forma en que el sistema cumplirá con los requisitos identificados durante la fase de análisis.

> **El diseño de software es el proceso de representar las funciones de cada sistema de software de manera tal que se pueda transformar con facilidad en uno o más programas de computación.**

Este proceso **comienza con la identificación de los reportes y salidas** que el sistema debe producir, luego se determinan con toda precisión los datos específicos para cada reporte y salida. El diseño también indica los datos de entrada, los cálculos intermedios y los que se almacenarán. Se describen en detalle los procedimientos de cálculo y los datos individuales. Se seleccionan las estructuras de archivo y los dispositivos de almacenamiento. Los procedimientos indican como procesar datos y producir salidas.

Esta **información detallada** del diseño se entrega al equipo de programación para comenzar el desarrollo de software. Durante la fase de programación, los diseñadores sirven de consultores a los programadores en cuanto a las dificultades que se puedan presentar al usar las especificaciones de diseño.

En esta tercera fase, los documentos de diseño del sistema y el software se preparan según el Documento de Requisitos de Software. Esto ayuda a definir la **arquitectura** general del sistema.

Hay dos tipos de documentos de diseño desarrollados en esta fase:

- Diseño de alto nivel (High level design - HLD)
    - Breve descripción y nombre de cada módulo.
    - Un resumen sobre la funcionalidad de cada módulo.
    - Relación de interfaz y dependencias entre módulos
    - Tablas de bases de datos identificadas junto con sus elementos clave
    - Diagramas de arquitectura completos junto con detalles tecnológicos.

- Diseño de bajo nivel (Low level design - LLD)
    - Lógica funcional de los módulos.
    - Tablas de base de datos, que incluyen tipo y tamaño
    - Detalle completo de la interfaz.
    - Aborda todos los tipos de problemas de dependencia
    - Listado de mensajes de error
    - Entradas y salidas completas para cada módulo.

Fase 4: Codificación

Una vez que finaliza la fase de diseño del sistema, la siguiente fase es la codificación. En esta fase, los desarrolladores comienzan a construir todo el sistema escribiendo código usando el lenguaje de programación elegido. En la fase de codificación, las tareas se dividen en unidades o módulos y se asignan a los distintos desarrolladores. Es la fase más larga del proceso del ciclo de vida del desarrollo de software.

En esta fase, el desarrollador debe seguir ciertas pautas de codificación predefinidas. También necesitan usar herramientas de programación como compilador, intérpretes, depurador para generar e implementar el código.

Los programadores son también **responsables de la documentación de los programas** y de dar explicaciones de cómo y porqué ciertos procedimientos se codifican de cierta forma.

Ya que el diseño se realiza como un conjunto de programas o unidades de programa, el programador escribe, en algún lenguaje ejecutable, un conjunto de unidades. También forma parte de sus responsabilidades efectuar la **Prueba de Unidades** que implica la comprobación de que cada unidad de programa cumple con su especificación.

Fase 5: Pruebas

Una vez que el software está completo y se implementa en el entorno de prueba. Se integran las unidades de programas individuales y el equipo de prueba comienza a probar la funcionalidad de todo el sistema. Esto se hace para **verificar** que toda la aplicación funciona de acuerdo con los requisitos del cliente.

Durante esta fase, el control de calidad y el equipo de prueba pueden encontrar algunos errores / defectos que comunican a los desarrolladores. El equipo de desarrollo corrige el error y lo envía de vuelta al control de calidad para una nueva prueba. Este proceso continúa hasta que el software esté libre de errores, estable y funcione de acuerdo con las necesidades comerciales de ese sistema.

Validación	¿Se ha construido el producto correcto? ¿Cumple con la definición de necesidades?
Verificación	¿Se ha construido correctamente el producto? Revisa si las funciones del producto son las que realmente quiere el cliente.

Es decir, el **mismo sistema se usa de manera experimental** para asegurarse que no tenga fallas y que funciona de acuerdo a las especificaciones y requisitos del usuario. Se alimentan conjuntos de datos de prueba como entrada y se examinan los resultados.

Se usan distintos grupos de prueba (ALFA y BETA) con la intención de descubrir fallas antes de que la organización o empresa instale el sistema y dependa de él.

Después de realizar las pruebas del sistema y sus correspondientes correcciones, el sistema de software se envía al cliente para que sea instalado.

La etapa de prueba representa **la última etapa de desarrollo dentro del ciclo de vida**.

Fase 6: Instalación e Implementación

Una vez que finaliza la fase de prueba de software y no quedan errores en el sistema, comienza el proceso de implementación final. Según los comentarios proporcionados por el gerente del proyecto, el software final se libera y se verifica si hay problemas de implementación.

La **instalación** es el proceso de verificar e instalar nuevos equipos, entrenar a los usuarios, instalar la aplicación y construir todos los archivos de datos necesarios para usarla.

Una vez instalado el software deberá sufrir varios cambios en el tiempo. En este sentido, **la implementación es un proceso en constante evolución**.

Fase 7: Mantenimiento

Una vez que se implementa el sistema, y los usuarios comienzan a usarlo, se producen las siguientes actividades.

Podemos clasificar de la siguiente manera, los **tipos o categorías de mantenimiento**:

- **Correctivo:** Se corrigen errores no detectados en fases previas.
- **Perfectivo:** Se agregan características nuevas al software existente.
- **Adaptativo:** Se actualiza el software existente teniendo en cuenta necesidades del contexto donde opera.
- **Preventivo:** Se modifican rutinas de código que son propensas a fallar.

La **evaluación** de un sistema se lleva a cabo para identificar puntos débiles y puntos fuertes. Esta evaluación ocurre a lo largo de cualquiera de las siguientes **dimensiones**:

- **Evaluación Operacional**: Cómo funciona el sistema, incluyendo su facilidad de uso, tiempo de respuesta, cuán adecuados son los formatos de información, confiabilidad global y nivel de utilización.

- **Impacto Organizacional**: Identificar y medir los beneficios para la organización de áreas tales como las finanzas, eficiencia operacional e impacto competitivo.

- **Opinión de los administradores**: Evaluación de los directivos y administradores dentro de la organización, así como de los usuarios finales.

- **Desempeño del desarrollo**: Evaluación del proceso de desarrollo de acuerdo a tiempo y esfuerzo de desarrollo, concordancia con los presupuestos y estándares y otros criterios de administración de proyectos. También incluye la valoración de los métodos y herramientas usados en el desarrollo.

Pequeño análisis de costos

Se puede hacer una tabla estimativa de los **costos relativos de desarrollo** de cada fase del ciclo de vida mencionado.

Tipos de Sistemas	Costo Porcentual por Fase		
	Análisis/Diseño	Implementación	Pruebas
Sistemas de Mandato y Control	46%	20%	34%
Sistemas Aéreos	34%	20%	**46%**
Sistemas Operativos	33%	17%	**50%**
Sistemas Científicos	44%	26%	30%
Sistemas Comerciales	44%	28%	**28%**

De esta tabla puede deducirse que **los costos de desarrollo de software son mayores al principio y al final del ciclo de desarrollo**. Esto dice que:

> La reducción de costos se logra mediante un diseño más efectivo del software y una verificación y validación más eficaces del tiempo de vida. Esto reducirá, finalmente, los costos de las pruebas.

Los costos de la fase final, **mantenimiento**, suelen superar a los costos de desarrollo por factores que varían de **dos a cuatro**. La mayor parte de los costos de mantenimiento no resultan de errores del sistema sino de **cambios en las necesidades**. Por lo tanto, para reducir los costos totales del ciclo de vida, debe establecerse una expresión, lo más exacta posible, de las necesidades reales del usuario.

Procesos del Ciclo de Vida

En 1987 la International Organization for Standarization (ISO) y la International Electrotechnical Commission (IEC) estableció la Joint Technical Committee (JTC1), la cual aprobó el estándar del **Software Life Cycle Processes**. El objetivo de la JTC1 es "La estandarización en el campo de los sistemas y equipos de tecnología de la información".

La Norma ISO-IEC 12207 se encarga de todo lo relacionado al ciclo de vida del software, desde la conceptualización de ideas hasta la retirada y consta de procesos para la adquisición y suministro de proyectos y servicios del software, estableciendo pautas para su control y mantenimiento.

Objetivos de la norma ISO-IEC 12207

El objetivo más importante de esta norma es proporcionar una estructura común para que los compradores, proveedores, desarrolladores, personal de mantenimiento, operadores, gestores y técnicos involucrados en el desarrollo de software utilicen un lenguaje en común.

Los objetivos de una empresa que busca la certificación con la norma ISO/IEC 12207:2008 son transportar a los clientes o socios la seguridad de que la empresa utiliza los procesos en relación con las prácticas de confianza de la industria. Además, los principios promovidos dentro de estas normas darán una plataforma sólida para administrar una solución de software desde sus inicios.

Características de la norma

- Proceso estructurado utilizando terminología aceptada.
- Documento relativamente de alto nivel
- No especifica detalladamente como realizar las actividades.
- No prescribe el nombre, el formato o el contenido de la documentación.

Alcances

El alcance de la norma es establecer un marco de referencia común para los procesos del ciclo de vida del software. Contiene procesos, actividades y tareas para aplicar durante el suministro, desarrollo, operación y mantenimiento de productos de software. Los procesos son descritos en términos de lograr los propósitos y salidas.

La norma no define cómo o en qué orden se lograrán los propósitos y salidas de los procesos. Los resultados serán alcanzados en una organización siguiendo prácticas detalladas para generar productos de trabajo. Estas prácticas realizadas y las características de los productos de trabajo son indicadores que demuestran si los propósitos específicos están siendo logrados. Además, la norma permite a una organización definir "como" un proceso será ejecutado conservando de esta forma la

flexibilidad necesaria para que los países o las organizaciones la implementen de acuerdo a la cultura local o a la tecnología disponible.

Aplicación

El estándar está diseñado para ser aplicado por una organización o por contrato entre dos o más organizaciones. Con el fin de facilitar la aplicación del estándar ya sea internamente o contractualmente, las tareas se expresan en el lenguaje contractual. Cuando se aplica internamente, el lenguaje contractual se interpreta como una tarea autoimpuesta.

El estándar es flexible y se puede utilizar con: cualquier modelo de ciclo de vida (como, cascada, evolutivo, espiral, u otro), cualquier método de ingeniería de software (diseño orientado a objetos, codificación estructurada, pruebas de arriba hacia abajo, o de otro tipo); o de los lenguajes (Ada, assembler, lenguaje máquina, u otro). Estos son muy dependientes tanto en el proyecto de software y el estado de la tecnología, y su selección se deja al usuario del estándar.

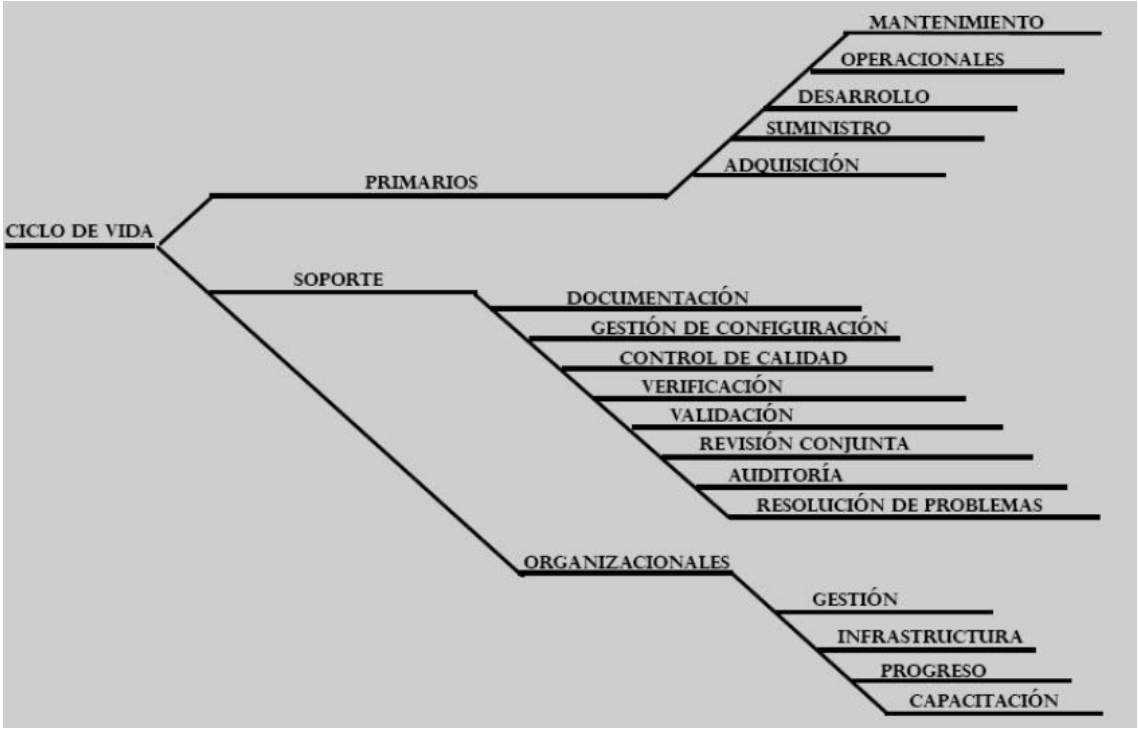
Limitaciones

El estándar no es un sustituto de la sistemática, disciplinada gestión e ingeniería de sistemas de software. La estándar se limita a establecer un marco donde los procesos, actividades y tareas relacionadas con el software pueden ser razonablemente identificadas, planificadas, y actuar en consecuencia.

Un punto clave a recordar es que la estándar sólo contiene un conjunto de bloques de construcción bien definidos (procesos); el usuario de la estándar debe seleccionar, adaptar, y montar estos procesos y sus actividades y tareas según sea apropiado y rentable para la organización y el proyecto.

Fases y Procesos

Esta norma agrupa las actividades que pueden llevarse a cabo durante el ciclo de vida del software en cinco procesos principales, ocho procesos de apoyo y cuatro procesos organizativos.



Cada **proceso** está diseñado en términos de sus propias **actividades**, cada una de las cuales está diseñado además en términos de las **tareas** que las componen.

El estándar aplica los **principios de gestión de calidad total**. Para empezar, el estándar trata todas las actividades, incluidas las relacionadas con la calidad, como una parte integral del ciclo de vida del software.

Los procesos primarios

Procesos de adquisición: Este proceso define las actividades y tareas del cliente que adquiere un producto de software o servicio por contrato, que puede ser el servicio completo o una parte de este.

El cliente presenta las necesidades de los usuarios, este proceso comienza con la definición de esta necesidad, continua con la preparación y emisión de una solicitud de propuesta, la selección de un proveedor y la gestión del proceso de adquisición a través de la aceptación del sistema. Entonces este proceso consta de las siguientes actividades, junto a las tareas específicas: Iniciación, solicitud de preparación de propuesta, elaboración y actualización del contrato, monitoreo del proveedor y, aceptación y finalización. Las 3 primeras se producen antes del acuerdo y las últimas 2 después del acuerdo.

Proceso de suministro: este proceso contiene las actividades y tareas del proveedor. Se compone de las siguientes actividades: Iniciación, preparación de la respuesta, contrato, planificación, ejecución y control, revisión y evaluación y, entrega y terminación.

Puede ser iniciado por la decisión de preparar una propuesta para responder a la petición de un cliente o mediante la firma de un acuerdo con el comprador para proporcionar un servicio. El servicio puede ser el desarrollo de un producto de software,

la operación de un sistema con un software o el mantenimiento de un producto. Luego de esto continua con la identificación de los procedimientos y los recursos necesarios para gestionar y asegurar el servicio, incluido el desarrollo y ejecución de los planes a través de la entrega al cliente.

Proceso de desarrollo: Este proceso del ciclo de vida contiene las actividades y tareas del desarrollador de software. El desarrollo a largo plazo denota tanto el desarrollo de nuevo software y modificación de un software existente. El proceso de desarrollo está destinado a ser empleado en al menos dos formas: (1) Como una metodología para el desarrollo de prototipos o para el estudio de los requisitos y el diseño de un producto o (2) Como un proceso para producir productos.

El proceso de desarrollo se compone de las siguientes actividades además de sus tareas específicas: implementación del proceso de análisis de requisitos del sistema, el diseño del sistema, análisis de requerimientos de software, diseño de la arquitectura del software, diseño detallado software, codificación y pruebas de software, integración del software, pruebas de calificación de software, integración de sistema, pruebas del sistema de calificación, instalación de software, y el apoyo de aceptación del software.

El orden de estas actividades no implica necesariamente un orden cronológico. Estas actividades pueden ser iteradas y solapadas. Todas las tareas en una actividad no necesitan ser completadas en la primera o cualquier iteración dada, pero deberían haberse completado cuando la iteración final llega a su fin. Estas actividades y tareas se pueden usar para la construcción de uno o más modelos de desarrollo (tales como, cascada, incremental, evolutivo, espiral, o de otro tipo, o una combinación de éstos) para un proyecto o una organización.

El estándar permite asentar las bases para los requisitos, diseño y código en los puntos predeterminados durante el desarrollo del producto. Lo cual inhibe los cambios prematuros o no planificados a estos requisitos y promueve el control de cambio efectivo. El estándar también proporciona los foros (es decir, la revisión conjunta y procesos de auditoría) para las partes interesadas en participar.

Proceso de operación: Este proceso de ciclo de vida, que forma parte de la operación total del sistema, contiene las actividades y tareas del operador de un sistema de software. El proceso comprende el funcionamiento del software y de apoyo operativo a los usuarios. Consta de las siguientes actividades, junto con sus tareas específicas: implementación del proceso, pruebas de funcionamiento, operación del sistema, y el apoyo del usuario.

Proceso de mantenimiento: Contiene las actividades y tareas del mantenedor. Este proceso se activa cuando un sistema se somete a modificaciones en el código y la documentación asociada debido a un error, una deficiencia, un problema, o la necesidad de una mejora o adaptación. El objetivo es modificar un sistema existente preservando al mismo tiempo su integridad. Cada vez que un producto de software necesita modificaciones, el proceso de desarrollo se invoca para efectuar y completar las modificaciones correctamente.

Este proceso consta de las siguientes actividades además de sus tareas específicas: implementación del proceso de análisis de problemas y modificaciones, aplicación de las modificaciones, revisión de mantenimiento/ aceptación, la migración, y la baja del software.

Los procesos de soporte

Este estándar contiene un conjunto de ocho procesos de apoyo. Un proceso de apoyo es compatible con cualquier otro proceso como una parte integral con un propósito distinto que contribuye al éxito y la calidad del proyecto. Un proceso de apoyo se invoca, según sea necesario, mediante la adquisición, suministro, desarrollo, operación o proceso de mantenimiento, o en otro proceso de apoyo.

Proceso de documentación: Este es un proceso para registrar la información producida por un proceso de ciclo de vida. El proceso define las actividades para planificar, diseñar, desarrollar, editar, distribuir y mantener los documentos necesarios por todos los interesados, tales como gerentes, ingenieros y usuarios del sistema. Las cuatro actividades junto con sus tareas son: implementación del proceso, diseño y desarrollo y, producción y mantenimiento.

Proceso de gestión de la configuración: Se emplea este proceso para identificar, definir, y alinear la base de los elementos de software en un sistema, para controlar las modificaciones y versiones de los elementos, para registrar e informar el estado de los elementos y las peticiones de modificación, para asegurar la integridad y exactitud de los elementos, y para controlar el almacenamiento, la manipulación y la entrega de los artículos. Este proceso consiste en: la ejecución de procesos, identificación de configuración, control de configuración, estado de la configuración, la evaluación de configuración, y gestión y administración de liberación.

Proceso de control de calidad: Este proceso proporciona el marco para asegurar la independencia y objetividad de conformidad de los productos o servicios con sus requisitos contractuales y la adhesión a sus planes establecidos. Para ser imparcial, el aseguramiento de la calidad debe tener libertad organizacional

frente a aquellas personas directamente responsables de desarrollar el producto de software o ejecutar los procesos del proyecto. Este proceso consiste en: la implementación de procesos, aseguramiento del producto, aseguramiento de procesos y aseguramiento de sistemas de calidad.

Proceso de verificación: Este proceso proporciona las evaluaciones relacionadas con la verificación de un producto o servicio de una actividad determinada. La verificación determina si los requisitos para un sistema son completos y correctos, y que los resultados de una actividad cumplan los requisitos y condiciones que se les imponen en las actividades anteriores. Abarca la verificación del proceso, de los requisitos, del diseño, del código, de la integración y la documentación. La verificación no alivia las evaluaciones asignadas a un proceso, al contrario, los complementa.

Proceso de validación: Determina si el sistema final cumple con su uso específico previsto. El alcance de la validación depende de la criticidad del proyecto. No sustituye a otras evaluaciones, sino que los completa.

Verificación o validación pueden ser realizadas por el que adquiere el producto, el proveedor o un tercero independiente. Cuando son ejecutados por una organización independiente del proveedor o desarrollador, se les llama proceso de verificación y validación independiente.

Proceso de revisión conjunta: El Proceso de Revisión Conjunta es un proceso en el cual se evalúan el estado y productos de una actividad de un proyecto. Las revisiones conjuntas se hacen tanto a nivel de administración de proyecto como a nivel técnico y se mantienen a lo largo de la vida del proyecto o contrato. Este proceso puede ser empleado por cualquiera de las dos partes, donde una (parte revisora) revisa a otra (parte revisada).

Proceso de auditoría: Este proceso proporciona el marco para las auditorías formales, establecidas en el contrato de los productos o servicios del proveedor. El auditor evalúa los productos del auditado y actividades con énfasis en el cumplimiento de los requisitos y planes. Una auditoría bien puede llevarse a cabo por la entidad adquirente.

Proceso de resolución de problemas: Este proceso proporciona el mecanismo para instituir un proceso de circuito cerrado para la resolución de problemas y tomar acciones de corrección para eliminar los problemas a medida que se detectan. Además, el proceso requiere la identificación y análisis de las causas y la reversión de las tendencias de los problemas reportados. El término "problema" incluye la no-conformidad.

Los procesos organizacionales

Este estándar contiene un conjunto de cuatro procesos de la organización. Una organización emplea un proceso organizativo para realizar funciones en él, a nivel corporativo de la organización, por lo general más allá o en los proyectos. Un proceso de organización puede apoyar cualquier otro proceso. Estos procesos ayudan a establecer, controlar y mejorar otros procesos.

Proceso de gestión: Este proceso define las actividades y tareas del gerente de un proceso de ciclo de vida del software, tales como el proceso de adquisición, proceso de suministro, proceso de operación, proceso de mantenimiento, o el proceso de apoyo. Las actividades abarcan: iniciación y definición del alcance, planificación, ejecución y control, revisión y evaluación, y cierre.

A pesar de que los procesos primarios, en general, tienen actividades de gestión similares, son lo suficientemente diferentes a nivel de detalle debido a sus diferentes metas, objetivos y métodos de operación.

Proceso de infraestructura: Este proceso define las actividades necesarias para el establecimiento y mantenimiento de una infraestructura subyacente para un proceso de ciclo de vida. Este proceso tiene las siguientes actividades: proceso de implementación, establecimiento de la infraestructura, y mantenimiento de la

infraestructura. La infraestructura puede incluir hardware, software, estándares, herramientas, técnicas, y las instalaciones.

Proceso de progreso: El estándar proporciona las actividades básicas al nivel superior que una organización necesita para evaluar, medir, controlar y mejorar el proceso de ciclo de vida. Las actividades comprenden: el establecimiento de procesos, evaluación de procesos y mejora de procesos. Las experiencias de la aplicación

de los procesos del ciclo de vida de los proyectos se utilizan para mejorar los procesos. Los objetivos son mejorar los procesos en beneficio de la organización en su conjunto y los proyectos actuales y futuros para el avance de las tecnologías de software en toda la organización.

Proceso de capacitación: Este proceso puede ser usado para identificar y hacer el suministro oportuno para adquirir o desarrollar los recursos y habilidades del personal en los niveles de gestión y técnicos. El proceso requiere que se elabore un plan de formación, se genere material de capacitación, y se brinde capacitación al personal en forma oportuna.

Ingeniería de Software

Concepto de Software

El software es:

- Alma y cerebro de una computadora.
- El conocimiento adquirido acerca de un área de aplicación.
- Conjunto de programas y datos necesarios para convertir una computadora (de propósito general) en una máquina de propósito específico para una aplicación particular.
- Documentación producida durante el desarrollo de un sistema software-intensivo.

Pensar que el software es sólo programas genera problemas: Ej. medir la productividad por líneas de código producidas por unidad de tiempo (capacidad de generar código productividad en la construcción del sistema). Así se condiciona el ambiente para producir código que lleva a montañas de código que no se pueden integrar para trabajar como un sistema y a la construcción de sistemas técnicamente correctos que no satisfacen las necesidades de los usuarios.

El software es aspectos de la información. El software no sólo son programas ejecutables, ya que se excluiría toda información relevante.

La clave de un desarrollo exitoso es no perder o alterar información introduciendo errores.

Características del Software

Características conceptuales:

- Objetivo de la Ingeniería: Construcción de productos.
- Para la Ingeniería de software, el producto son los sistemas de software.
- El software es maleable.
- Se cree que cambios en el software son fáciles.
- Un cambio en el software debe verse como un cambio en el diseño más que en el código.
- Su producción es humano-intensiva: requiere más ingeniería que manufactura. El proceso de producción de software se vincula más con el diseño e implementación que con la manufactura.
- En la Ingeniería de Software, a diferencia de lo tradicional, el ingeniero dispone de herramientas para describir el producto que no son distintas del producto.
- A menudo, las cualidades del producto están entremezcladas en especificaciones con las cualidades del diseño.

Cualidades del Software según la visión del:

  - Usuario:

        - Confiable
        - Eficiente
        - Fácil de usar
    - Productor:
        - Verificable
        - Mantenible
        - Portable
        - Extensible
    - Proyect Manager (Proceso):
        - Productivo
        - Fácil de Controlar

Cualidades externas: Visibles para los usuarios.

Cualidades internas: Sólo visible para los desarrolladores.

Existe una fuerte relación entre ellas: La cualidad interna de la verificabilidad, se requiere para alcanzar la cualidad externa de la confiabilidad.

Confiabilidad >>> verificabilidad

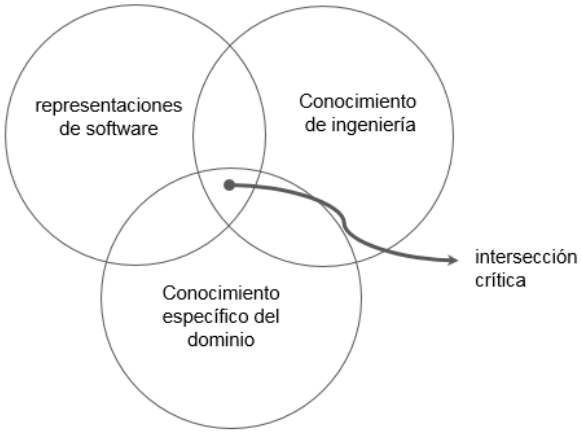
Cualidades del software (Procesos y Productos)

- Corrección funcional
- Confiabilidad
- Robustez
- Performance
- Amigabilidad
- Verificabilidad
- Mantenibilidad
- Reusabilidad
- Portabilidad
- Comprensibilidad
- Interoperabilidad
- Productividad
- Oportunidad
- Visibilidad

Clases de Información

Clases de Información:

- Representaciones de software
- Conocimiento de software (información de desarrollo)
- Conocimiento del dominio específico (información de la aplicación)



Representaciones del Software

Cualquier información que en forma directa representa un eventual conjunto de programas y los datos asociados:

- Programas
- Diseños detallados
- Diseño de arquitectura (Diagramas de Estructuras)
- Especificaciones escritas en lenguaje formal
- Requisitos del sistema expresados en una combinación de notaciones.
- Etc.

Conocimiento de la Ingeniería de Software

Toda la información relativa al desarrollo en general (ej.: cómo usar un método específico de diseño) o relativa a un desarrollo particular (ej.: programa de testeo en un proyecto).

- Información relativa al proyecto.
- Información sobre la tecnología de software (métodos, conceptos, técnicas).
- Conocimiento acerca de sistemas similares.
- Información detallada relativa a la identificación y solución de problemas técnicos del sistema en desarrollo.

Conocimiento específico del dominio

Es esencial para la creación del software. Descubrirlo y ponerlo en práctica en forma útil es la esfera de un especialista en el área de aplicación.

- Conocimiento del proceso específico a ser controlado.
- Reglas de contabilidad.
- Procedimientos para actualizar y cambiar los registros de los empleados.
- Etc.

Formas que toma el software:

- Programas en lenguaje de máquina.

- Programas en lenguaje de alto nivel.
- Especificaciones.
- Necesidades.
- Requerimientos.
- Diseños arquitectónicos.
- Diseños detallados.
- Formatos de datos.
- Colecciones de programas.
- Programas a testear.
- Programas terminados.
- Sistemas en uso para producción

Otras formas:

- Análisis de Requisitos
- Documentación del usuario.
- Documentación de mantenimiento.
- Pedidos de cambio.
- Especificaciones de modificaciones.
- Informes de errores.
- Mediciones de performance.

El software es tanto un producto como un objeto, esto es: **conocimiento empaquetado**. Los programas (final de una cadena de representaciones que llamamos software) **contienen conocimiento**. Una de las principales razones de la reusabilidad es no perder este conocimiento.

Software como conocimiento: Balancear actividades de análisis y la construcción de programas. Es la visión más idónea para entender la naturaleza esencial de la actividad.

Software como producto: Se vincula con la organización del desarrollo de software.

Software como una serie de transformaciones: Provisión de herramientas para ayudar al desarrollo de software.

Productos de Software

Programas de Computadoras + Procedimientos + Documentación Asociada + Datos de la operación del sistema = **Conocimiento Acumulado**

Definición de Ingeniería de Software

Ya que actualmente las economías personales, empresariales, nacionales e internacionales dependen cada vez más de las computadoras y sus sistemas de software, es que la práctica de la Ingeniería de Software tiene por objeto la construcción de grandes y complejos sistemas en forma rentable.

Los problemas que se presentan en la construcción de grandes sistemas de software no son simples versiones a gran escala de los problemas de escribir pequeños programas en computadoras.

La complejidad de los programas pequeños es tal que una persona puede comprender con facilidad y retener en su mente todos los detalles de diseño y construcción. Las especificaciones pueden ser informales y el efecto de las modificaciones puede ser evidente de inmediato. Por otro lado, los grandes sistemas son tan complejos que resulta imposible para cualquier individuo recordar los detalles de cada aspecto del proyecto. Se necesitan técnicas más formales de especificación y diseño: debe documentarse adecuadamente cada etapa del proyecto y es esencial una cuidadosa administración.

El término Ingeniería de Software se introduce a fines de la década del 60, al producirse la llamada "Crisis del Software" a causa de la aparición de Hardware de 3ª generación. Estas nuevas máquinas eran de una capacidad muy superior a las máquinas más potentes de 2ª generación y su potencia hizo posible aplicaciones hasta entonces irrealizables.

Las primeras experiencias en la construcción de grandes sistemas de software mostraron que las técnicas de desarrollo hasta entonces conocidas eran inadecuadas. Entonces se presentó la urgente necesidad de nuevas técnicas y metodologías que permitieran controlar la complejidad inherente a los grandes sistemas de software.

Ingeniería de Software = Programming in the Large

Entonces, convenimos en definir:

Ingeniería de Software: Disciplina que trata de la construcción de grandes sistemas que no puede manejar un único individuo. Usa principios metodológicos de la Ingeniería para el desarrollo de sistemas, es decir, sistematiza el desarrollo de sistemas. Adquiere el nivel de una disciplina. Consta de aspectos técnicos y aspectos no técnicos.

El Ingeniero de Software debe tener profundos conocimientos de las técnicas de computación y debe poder comunicarse en forma oral y escrita. Debe comprender los problemas de administración de proyectos relacionados con la producción de software y debe poder apreciar los problemas de los usuarios del software cuando no lo entienden.

Definiciones de IS:

IEEE

El uso de métodos sistemáticos, disciplinados y cuantificables para el desarrollo, operación y mantenimiento de software. (Es decir, la aplicación de prácticas de Ingeniería de Software). O bien, el estudio de técnicas relacionadas con lo anterior.

Fairley

Disciplina tecnológica y de administración que se ocupa de la producción y evolución sistemática de productos de software que son desarrollados y modificados dentro de los tiempos y costos estimados.

Ghezzi

Campo de la ciencia de la computación que trata con la construcción de sistemas de Software que son tan grandes o complejos que son construidos por un equipo o equipos de ingenieros.

Conocimientos requeridos:

- Principios teóricos de representación y computación.
- Aplicación de métodos formales.
- Uso de notaciones de modelización, especificación, diseño y programación.
- Combinación de conocimientos de:
- Metodologías.
- Tecnologías.
- Técnicas de administración de proyectos

Ingeniería de software: Disciplina que se encarga de planificar, desarrollar y mantener software construido por un equipo o equipos de ingenieros, de forma sustentada, ordenada y cuantificable, ajustándose a costos y tiempo previamente estimados (de forma rentable).

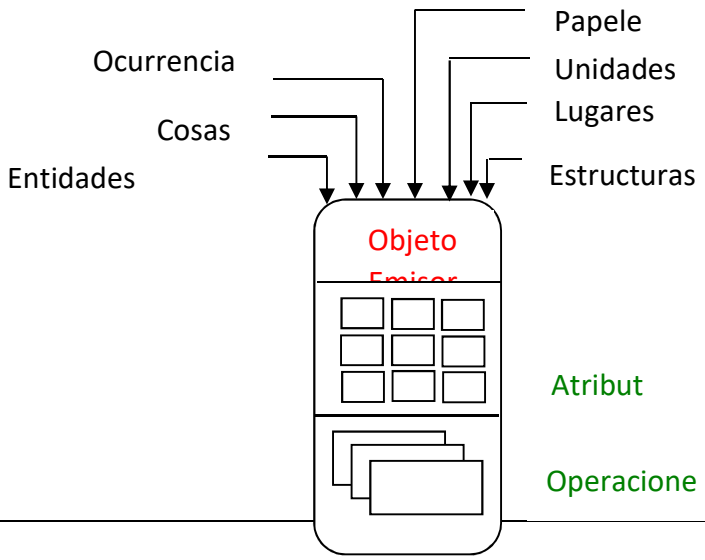
Modelo de Análisis y Diseño Orientado a Objetos (Modelo ADOO)

Identificación de Objetos

Daremos a continuación una serie de directrices informales que nos ayudarán en la identificación de clases de objetos, atributos, operaciones y mensajes de un modelo de objetos.

- Identificación de clases y objetos: Los objetos físicos que uno observa en una habitación son fáciles de identificar. Pero esto no ocurre cuando uno observa el espacio de un problema en una aplicación de software. Se comienza a identificar analizando gramaticalmente la narrativa de procesamiento, subrayando los nombres o cláusula nominal, los sinónimos deben destacarse. Si se requiere que el objeto implemente una solución, éste formará parte del **espacio de solución**; pero si el objeto se necesita solamente para describir una solución, ésta forma parte del **espacio del problema**.
  Los objetos pueden ser:

  - **Entidades externas**: productores o consumidores de información a usar por un sistema computacional (ej.: otros sistemas, personas, etc).
  - **Cosas**: que son parte del dominio de información del problema (ej.: informes, presentaciones, cartas, señales).
  - **Ocurrencias o eventos**: que ocurren dentro del contexto de operación del sistema (ej.: transferencia de una propiedad o la terminación de una serie de movimientos de un robot).
  - **Papeles o roles**: desempeñados por personas que interactúan con el sistema (ej.: Director, ingeniero, vendedor, cliente, etc)
  - **Unidades organizacionales**: que son relevantes en una aplicación (ej.: división, grupo, equipo).
  - **Lugares**: que establece el contexto del problema y la función general del sistema (ej.: planta de producción o muelle de carga).
  - **Estructuras**: que definen una clase de objetos o clases relacionadas de objetos (ej.: sensores, vehículos de cuatro ruedas o computadoras).



El software HogarSeguro le permite al propietario de la casa configurar el sistema de seguridad una vez que este se instala, controla todos los sensores conectados al sistema de seguridad, e interactúa con el propietario a través de un teclado numérico y teclas de función contenidas en el panel de control de HogarSeguro mostrado en la figura anterior.

Durante la instalación, el panel de control de HogarSeguro se usa para "programar" y configurar el sistema. A cada sensor se le asigna un número y tipo, se programa una contraseña maestra para activar y desactivar el sistema, y se introducen números de teléfono para marcar cuando un evento sea detectado por un sensor.

Cuando se reconoce un evento de sensor, el software invoca una alarma audible asociada al sistema. Después de un tiempo de espera especificado por el propietario durante las actividades de configuración del sistema, el software marca un número de teléfono de un servicio de control, proporciona información acerca de la localización, e informa de la naturaleza del evento detectado. El número será remarcado cada 20 segundos hasta obtener una conexión telefónica.

Toda la interacción con HogarSeguro está gestionada por un subsistema de interacción con el usuario que toma la entrada a partir del teclado numérico y teclas de función, y muestra mensajes y el estado del sistema en la pantalla LCD. La interacción con el teclado toma la siguiente forma...

En general, un objeto nunca debe tener un nombre procedimental imperativo (ej.: inversión de imagen) ya que imagen podría ser un objeto pero la inversión es una operación que se aplica al objeto. El objetivo de la orientación a objetos es encapsular, pero manteniendo separados datos y operaciones sobre esos datos. En el ejemplo de HogarSeguro, si analizamos la narrativa de procesamiento, podemos proponer la siguiente lista de objetos potenciales:

objeto/clase potencial	clasificación general
Propietario	Papel(rol) o entidad externa
Sensor	Entidad externa
Panel de control	Entidad externa
Instalación	Ocurrencia
Sistema(sinónimo:sistema de seguridad)	Cosa
Número, tipo	No son objetos, son atributos de sensor

Contraseña maestra---------- Cosa

Número de teléfono---------- Cosa

Evento de sensor---------- Ocurrencia

Alarma audible---------- Entidad externa

Servicio de control---------- Unidad organizacional o entidad externa

La lista anterior continúa hasta que se hayan considerado Todos los nombres de la narrativa de procesamiento.

Como esta lista es de potenciales, se debe, de acuerdo a algún criterio, tomar una decisión final.

Coad y Yourdon sugieren seis características de selección para determinar si un objeto se incluye o no en el modelo de análisis.

1. **Información retenida**: (Need remembrance). El objeto será de utilidad durante el análisis solamente si la información acerca de él debe recordarse para que el sistema funcione.(¿Se necesita recordar sus potenciales atributos?). Puede ser que muchos objetos o clases del mundo real sean interesantes y surjan en las discusiones con el cliente, pero que no resulten ser relevantes para el sistema en consideración.
2. **Servicios necesarios**: (Needed Behavior). ¿El objeto, necesariamente, debe proveer algún comportamiento?. Un objeto potencial, cuya información deba ser retenida, debe poseer un conjunto de operaciones identificables que puedan cambiar de alguna manera el valor de sus atributos y que permitan, por lo menos, crearlo, conectarlo, accederlo y liberarlo.
3. **Atributos múltiples**(usualmente): (usually múltiple attributes). Esta característica sirve como un filtro muy fino. Los objetos potenciales con un único atributo seguramente serán mejor presentados como un atributo de otro objeto durante el análisis.
4. **Atributos comunes**: (Always-applicable attributes).¿Se puede identificar un conjunto de atributos que son aplicables a cada objeto de la clase?. Si algunos objetos de la clase son tales que no se aplican todos los atributos de la misma, quizás sea indicada una estructura de generalización o especialización.
5. **Operaciones comunes**: (Always-applicable services). ¿Se puede identificar comportamientos aplicables a todas y cada ocurrencia del objeto en la clase?.Los servicios pueden ser algoritmos simples o complejos, pero deberían poder aplicarse a todos los objetos de la clase.
6. **Requisitos esenciales**: (Domain-based requirements). Son requisitos que el sistema debe tener independientemente de la tecnología usada para

construirlo. Las entidades externas que aparecen en el espacio del problema y producen o consumen información esencial para la producción de cualquier solución para el sistema serán casi siempre definidas como objetos en el modelo de requisitos.

Para ser considerado un objeto válido a incluir en el modelo de requisitos, un objeto potencial debe satisfacer todas (o casi todas) las características anteriores. Esta decisión es algo subjetiva y una evolución posterior puede llegar a descartar o a recluir un objeto.

El primer paso del AOO debe ser la definición de objetos y la consiguiente toma de decisiones.

Debe mostrarse que: (1) La lista de objetos potenciales debe incluir todos los objetos de la narrativa (2) algunos de los objetos rechazados pueden ser atributos de los objetos aceptados (EJ. : contraseña maestra y nº de teléfono atributos del sistema). Y (3) diferentes descripciones del problema pueden provocar aceptaciones o rechazos diferentes (EJ. si el propietario fuese identificado por su voz cumpliría 1 y 2 y hubiese sido aceptado).

Objeto/clase potencial	Característica aplicable
Propietario----------	Rechazado: fallan 1 y 2 aunque 66 es aplicable
Sensor----------	Aceptado: son todas aplicables
Panel de Control----------	Aceptado: son todas aplicables
Instalación----------	Rechazada
Sistema(sinónimo:Sistema de Seguridad)---	Aceptado: son todas aplicables
Número, tipo----------	Rechazado: falla 3. Son atributos de sensor
Contraseña maestra----------	Rechazado: falla 3
Número de teléfono----------	Rechazado: falla 3

Evento de sensor----------	Aceptado: son todas aplicables
Alarma audible----------	Aceptado: 2, 3, 4, 5 y 6 aplicables
Servicio de control----------	Rechazado: fallan 1, 2 aunque 6 es aplicable

Especificación de atributos

Los atributos describen un objeto que ha sido seleccionado para ser incluido en el modelo de análisis. Esencialmente, aclaran el significado del objeto en el contexto del espacio del problema. El contexto del problema que se está analizando define, en gran medida, los atributos del objeto. Por ejemplo, en un sistema que lleve estadísticas de jugadores profesionales de baseball, los atributos del objeto jugador serían muy diferentes de los atributos del mismo objeto cuando se use dentro del contexto del sistema de pensiones para jugadores profesionales.

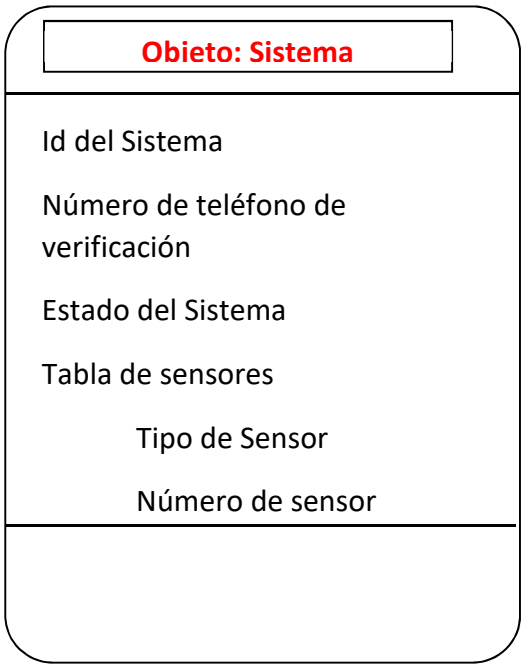
Para desarrollar un conjunto significativo de atributos para un objeto, el analista debe seleccionar de la narrativa de procesamiento aquellos **elementos que "pertenecen" al objeto**. Además, debe responderse a la siguiente pregunta. ¿Qué elementos (compuestos y /o simples) definen completamente al objeto en el contexto del problema actual?

Consideremos, a modo de ejemplo, el objeto Sistema definido para HogarSeguro. El propietario puede configurar el sistema de seguridad para reflejar información de los sensores, sobre la respuesta de la alarma, sobre la activación / desactivación, sobre identificación, etc.

Usando la notación del DD podríamos representar estos elementos de datos compuestos de la siguiente manera:

Información de sensores	=	Tipo de sensor	+	Número de sensor	+	Umbral de alarma
Información de respuesta de la alarma	=	Tiempo De retardo	+	Número de teléfono	+	Tipo de alarma
Información de activación / desactivación	=	Contraseña maestra	+	Cantidad de intentos permitidos	+	Contraseña temporal
Información de identificación	=	ID. del sistema	+	Verificación del número de teléfono	+	estado del sistema

Cada uno de los elementos de datos a la derecha del signo = puede volver a definirse en un nivel elemental. Así, tenemos una lista razonable de atributos para el objeto sistema.



Definición de operaciones

Las operaciones definen el comportamiento de un objeto y cambian, de alguna manera, los atributos de dicho objeto. Más concretamente, una operación cambia valores de uno o más atributos del objeto y por lo tanto deben tener "conocimiento" de la naturaleza de esos atributos y deben estar implementadas de manera tal que permita manipular las estructuras de datos que han sido derivadas de dichos atributos.

Las operaciones se pueden clasificar en tres grandes categorías:

- **Operaciones que manipulan datos.**
- **Operaciones que realizan cálculos.**
- **Operaciones que monitorizan un objeto frente a algún suceso de control**

La selección de operaciones se hace aislando los verbos de la narrativa de procesamiento, por ejemplo:

"al sensor se le asigna un número y un tipo".

ó

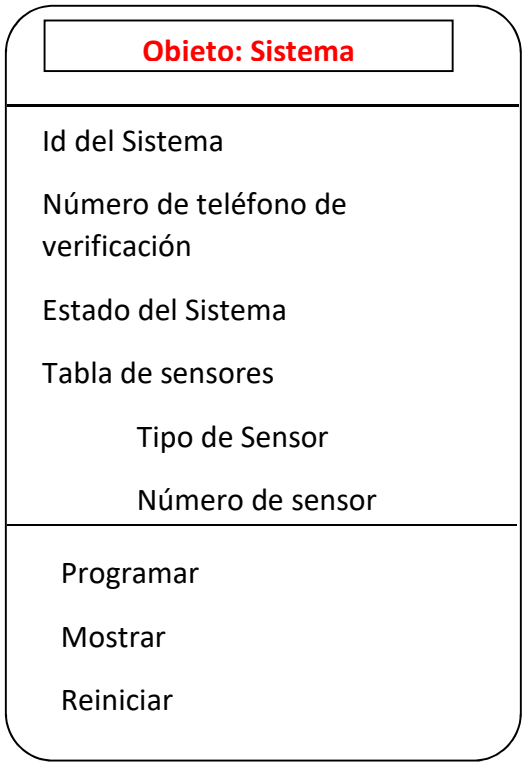
"se programa una contraseña maestra para activar / desactivar el sistema".

Estas frases indican que:

- Una operación de "asignación" es relevante para el objeto sensor.
- Una operación "programar" se aplicará al objeto sistema. En una investigación más detallada "programar" podría estar dividida en varias sub-operaciones más específicas requeridas para configurar el sistema: especificar números de teléfono, crear tablas de sensores, introducir la (s) contraseña(s), etc.
- Activar y desactivar son operaciones aplicables al sistema. El estado del sistema puede definirse:

Estado del sistema = [activado | desactivado]

También podemos ganar información interna extra en otras operaciones si consideramos la **comunicación** que ocurre entre objetos, ya que esta comunicación implica un conjunto de operaciones que deben estar presentes.



Fin de la definición del objeto

La definición de operaciones es el último paso para completar la especificación del objeto.

Las operaciones se eligen a partir de la narrativa de procesamiento y las operaciones adicionales se añaden considerando la "historia de la vida" de un objeto y los mensajes que se pasan entre objetos definidos por el sistema.

La **historia de la vida genérica** de un objeto puede definirse reconociendo que el objeto debe crearse, manipularse, modificarse y borrarse. Esto puede expandirse para reflejar actividades conocidas que ocurren durante su tiempo de vida (ej.: mientras HogarSeguro se mantiene operativo).

Otras operaciones se determinan a partir de las **comunicaciones** semejantes entre objetos: **evento sensor** mandará un mensaje a **sistema** para **mostrar** en pantalla la localización y número de evento; el **panel de control** enviará un mensaje de **reinicialización** para actualizar el estado del sistema; **alarma audible** enviará un mensaje de **consulta**; panel de control enviará un mensaje de **modificación** para cambiar uno o más atributos sin reconfigurar el **sistema; evento sensor** enviará un mensaje para **llamar** al número(s) de teléfono(s) contenido(s) en el objeto.

Un enfoque similar será usado para cada uno de los objetos definidos por **HogarSeguro**. Luego de definir atributos y operaciones para todos los objetos especificados Hasta Ahora, se han creado los inicios del **modelo AOO.**

Casos de Uso (Escenarios ó Escenas de uso)

En cualquier actividad de análisis de software, el primer paso es siempre la **recopilación de requisitos**, que puede tomar la forma de encuentros entre el analista y el cliente para fijar los requisitos básicos del sistema y del software.

Basándose en estos requisitos, el analista puede crear un conjunto de escenarios de tal manera que cada uno identifique una parte (hilo) del uso que se le dará al sistema a construir. Es decir, los escenarios, o **casos de uso**, aportan una descripción acerca de cómo el sistema será usado.

Para crear un caso de uso, el analista debe primero identificar los actores:

Actores: cualquier cosa que se comunique con el sistema o producto y que sea externo a él.

Tipos de personas o dispositivos que usan el sistema o producto.

Representan papeles ejecutados por personas o dispositivos cuando el sistema está en operación.

Se debe notar que un actor y un usuario no son la misma cosa. Un usuario típico puede desempeñar cierto número de roles, mientras que un actor representa una clase de entidades externas (a menudo, las personas) que sólo desempeñan un único papel cuando usan el sistema.

Como en otros aspectos del modelo AOO, es posible identificar **actores primarios** durante la primera iteración y **actores secundarios** al aprender más sobre el sistema. Los actores primarios interactúan para lograr el funcionamiento requerido del sistema, y obtener de él el beneficio propuesto. Ellos trabajan directa y frecuentemente con el software.

Los actores secundarios existen para dar soporte al sistema de manera tal que los primarios puedan realizar su trabajo.

Una vez que se han identificado los actores primarios pueden desarrollarse los casos de uso, que describen la forma en la que el actor interactúa con el sistema.

Según **Jacobson** (92) el caso de uso debe responder a las siguientes cuestiones:

- ¿Cuáles son las tareas o funciones principales a realizar por el actor?
- ¿Qué información del sistema adquirirá, producirá o cambiará el actor?
- ¿Tendrá el actor que informar al sistema acerca de cambios en el entorno exterior?
- ¿Qué información desea el actor sobre el sistema?
- ¿Desea el actor ser informado sobre cambios inesperados?

En general, un caso de uso es simplemente una narración escrita que describe el papel de una actor al ocurrir la interacción con el sistema.

Ejemplo: Hogar Seguro

Podemos definir tres actores:

- El propietario (usuario).
- Los sensores (dispositivos adjuntos al sistema).
- El subsistema de monitorización y respuesta (la estación central que monitoriza a Hogar Seguro).

El actor propietario interactúa con el producto en un número de formas diferentes:

- Introduce una contraseña para permitir el resto de las interacciones.
- Interroga acerca del estado de una zona de seguridad.
- Interroga acerca del estado de un sensor.
- Presiona el botón de pánico en una emergencia.
- Activa/Desactiva el sistema de seguridad.

Un caso de uso para la activación del sistema sería:

El propietario observa el panel de control de Hogar Seguro para ver si está listo para las entradas. Si el sistema no está listo implica que un sensor (puerta o ventana) está abierto y el propietario deberá, físicamente, cerrarlo para que el indicador de disponibilidad esté presente.

El propietario usa el teclado numérico para ingresar una contraseña de cuatro dígitos, que se comparará con la contraseña válida almacenada en el sistema.

Si ésta es incorrecta, el panel de control emitirá un pitido una vez y se restaurará en espera de entradas adicionales. Si la contraseña es correcta, el panel de control esperará para otras acciones.

El propietario selecciona y escribe "fuera" o "en casa" para activar el sistema. El "en casa" activa solamente los sensores de perímetro (se desactivan los sensores internos detectores de movimiento). El "fuera" activa todos los sensores.

Cuando ocurre la activación, el propietario podrá observar una luz roja de alarma.

De manera similar se desarrollarán los casos de uso para otras interacciones del propietario. Cada caso de uso debe revisarse cuidadosamente. Si algún elemento de la interacción es ambiguo, es probable que una revisión del caso de uso indicará un problema.

Cada caso de uso aporta un escenario no ambiguo de interacción entre un actor y el software.

Los casos de uso también pueden usarse para especificar requisitos de tiempo u otras restricciones para el escenario.

Por ejemplo, se puede añadir al caso de uso anterior que los requisitos indican que la activación ocurre 30 segundos después de presionar la tecla "fuera" o "en casa".

Modelado CRC (Tarjetas Índice)

Una vez que se han desarrollado los escenarios de uso básicos para el sistema, es tiempo de identificar las **clases candidatas**, indicar sus **responsabilidades** y **colaboraciones**.

El modelado CRC aporta un medio sencillo de identificar y organizar las clases que resulten relevantes al sistema o requisitos del producto.

> Un modelo CRC es realmente una colección de tarjetas índice estándar que representan clases. Las tarjetas están divididas en tres secciones: a) una cabecera con el nombre de la clase, b) en el cuerpo, a la izquierda, las responsabilidades de la clase y c) en el cuerpo, a la derecha, los colaboradores.

El objetivo del modelo CRC, ya sea que use tarjetas índice virtuales o actuales, es desarrollar una representación organizada de las clases.

Las **responsabilidades** son los atributos y operaciones relevantes para la clase: "cualquier cosa que conoce o hace la clase".

Los **colaboradores** son aquellas clases necesarias para proveer a una clase con la información necesaria para completar una responsabilidad. En general, una colaboración implica una solicitud de información o una solicitud de alguna acción.

Clases

A la taxonomía de tipos de clases dada anteriormente: entidades externas, cosas, ocurrencia o eventos, roles, unidades organizacionales, lugares o estructuras; Firesmith (93) agrega las siguientes:

- **Clases dispositivo:** Modelan entidades externas tales como sensores, motores y teclados.
- **Clases propiedad**: Representan alguna propiedad importante del entorno del problema (ej.: establecimiento de créditos en el contexto de una aplicación de préstamos hipotecarios).
- **Clases interacción**: Modelan interacciones que ocurren entre otros objetos (ej.: una adquisición o una licencia).

Un objeto potencial debe satisfacer las seis características de selección para ser considerado como un posible miembro del modelo CRC.

Adicionalmente, los objetos y clases pueden clasificarse por un conjunto de características:

- **Tangibilidad:** ¿la clase representa algo tangible o representa información más abstracta?. (ej.: un teclado o sensor o una salida).
- **Inclusividad**: ¿la clase es atómica (no incluye otras clases) o agregada? (Incluye al menos un objeto anidado).
- **Secuencialidad**: ¿la clase es concurrente (posee su propio hilo de control) o secuencial? (es controlada por recursos externos).
- **Persistencia**: ¿es una clase transitoria (creada y eliminada durante la ejecución del programa), temporal (creada durante la ejecución del programa y eliminada cuando éste termina) o permanente? (es almacenada en una base de datos).
- **Integridad**: ¿es la clase corrompible (no protege sus recursos de influencias externas) o es segura (refuerza los controles de accesos a sus recursos)?

Con estas características, las tarjetas índices creadas como parte del modelo CRC pueden extenderse para incluir el tipo de la clase y sus características:

Nombre de la clase:	
Tipo de la clase: (dispositivo, propiedad, rol,...)	
Características de la clase: (tangible, atómica, concurrente,...)	
Responsabilidades	Colaboradores

Responsabilidades

Las pautas básicas para identificar responsabilidades: **atributos** y **operaciones** ya se expusieron y son:

Atributos: representan características estables de una clase, es decir, información sobre la clase que debe retenerse para llevar a cabo los objetivos del software especificados por el cliente. Pueden extraerse a partir del planeamiento de alcance o discriminarse a partir de la comprensión de la naturaleza de la clase.

Operaciones: cada operación elegida exhibe un comportamiento de la clase. Se extraen haciendo un análisis gramatical de la narrativa de procesamiento del sistema y extrayendo los verbos que son candidatos a transformarse en operaciones.

Wirfs-Brock y sus colegas (90) sugieren 5 pautas para especificar responsabilidades para la clase:

1. **La inteligencia del sistema debe distribuirse de manera igualitaria.**

   Se considera que la "inteligencia" de un sistema denota lo que el sistema sabe y lo que el sistema podría hacer.

   Esta inteligencia podría distribuirse de manera tal que las clases "tontas" (clases con pocas responsabilidades) actúen como sirvientes de unas pocas clases "listas", enfoque que hace que el flujo de control dentro del sistema sea claro, pero hace que los cambios sean más difíciles ya que concentra la inteligencia en unas pocas clases y aumenta el esfuerzo de desarrollo pues se necesitan más clases.

   La distribución igualitaria de inteligencia entre las clases, por el contrario, incrementa la **cohesión** del sistema (cada objeto conoce y actúa sobre algunos pocos elementos bien definidos y claros) y los **efectos laterales** provocados por cambios tienden a amortiguarse (la inteligencia del sistema se ha descompuesto entre muchos objetos).

   Para determinar si la inteligencia está distribuida equitativamente, se analiza la lista de responsabilidades de la clase. Si es demasiado larga, en relación a la de otras clases, esto indica una concentración de inteligencia.

   Además, las responsabilidades de cada clase deben mostrar el mismo nivel de abstracción (y complejidad).

2. **Cada responsabilidad debe establecerse lo más general posible.**

   Las responsabilidades (atributos y operaciones) deben residir en la parte alta de la jerarquía de clases.

   Como son genéricas, se aplicarán a todas las subclases.

   Adicionalmente, se puede usar polimorfismo para redefinir operaciones en las subclases.

3. **La información y el comportamiento asociado a ella, debe encontrarse dentro de la misma clase.**

   Esto implementa el principio de **encapsulamiento**, ya que los datos y procesos que los manipulan deben empaquetarse como una unidad cohesionada.

4. **La información sobre un elemento debe estar localizada dentro de una clase, no distribuida a través de varias clases.**

   Si la información está distribuida, el software se torna más difícil de mantener y probar. Una clase simple debe asumir la responsabilidad de almacenamiento y manipulación de un tipo específico de información, y no compartirla, de manera general, con otras clases.

5. **Compartir responsabilidades entre clases relacionadas cuando sea apropiado.**

Existen muchos casos en los cuales una gran variedad de objetos exhibe el mismo comportamiento, al mismo tiempo (ej.: los objetos jugador, cuerpo_del_jugador, brazos_del_jugador y cabeza_del_jugador en un juego de vídeo tienen sus propios atributos, pero comparten las responsabilidades de actualizar y visualizar cuando se acciona la palanca de mando (Joystick)).

Colaboradores

Las clases cumplen con sus responsabilidades en una de dos maneras:

1. Una clase puede usar sus propias operaciones para manipular sus propios atributos, cumpliendo con una responsabilidad particular.
2. Una clase puede colaborar con otras clases.

Wirfs-Brock y sus colegas (90) definen las colaboraciones de la siguiente manera:

> Las colaboraciones representan solicitudes de un cliente a un servidor en el cumplimiento de una responsabilidad del cliente... Una colaboración es la realización de un contrato entre el cliente y el servidor.
> Un objeto colabora con otro, si para ejecutar una responsabilidad, necesita enviarle un mensaje.
> Una colaboración simple fluye en una dirección, representando una solicitud del cliente al servidor.
> Desde el punto de vista del cliente, cada una de sus colaboraciones está asociada con una responsabilidad particular implementado por el servidor.

Las colaboraciones identifican relaciones entre clases. Cuando todo un conjunto de clases colabora para satisfacer algún requisito, es posible organizarlas en un subsistema.

Las colaboraciones se identifican determinando si una clase puede satisfacer cada responsabilidad. SI no puede, entonces necesita interactuar con otra clase (una colaboración). (Por ejemplo, panel de control debe trabajar en colaboración con el objeto sensor para que la responsabilidad *determinar_estado_del_sensor* pueda ejecutarse).

Para ayudar en la identificación de colaboradores, el analista puede examinar tres relaciones genéricas diferentes entre clases:

1. **La relación es-parte-de**: todas las clases que forman parte de una clase agregada están conectadas a ella a través de esta relación. (ej.: las clases cuerpo_de_jugador, brazos_del_jugador y cabeza_del_jugador es-parte-de la clase jugador).

2. **La relación <u>tiene-conocimiento-sobre</u>**: este tipo de relación se establece cuando una clase debe tener información sobre otra. (ej.: la responsabildad determinar_estado_del_sensor es un ejemplo de este tipo de relación).
3. **La relación <u>depende-de</u>**: implica que dos clases poseen una dependencia no realizable a través de las dos relaciones anteriores (ej.: la cabeza_del_jugador depende – de cuerpo_del_jugador para obtener la posición_central del jugador).

En todos los casos, el nombre de la clase colaboradora se registra en la tarjeta índice del modelo CRC al lado de la responsabilidad que genera dicha colaboración. Así, cada tarjeta índice tiene una lista de responsabilidades y de colaboraciones correspondientes que posibilitan la realización de las responsabilidades.

Validación

Cuando se ha desarrollado un modelo CRC, los representantes del cliente y de las organizaciones de ingeniería de software pueden recorrer el modelo usando el siguiente enfoque:

I. A todos los participantes de la revisión del modelo CRC se les da un subconjunto de las tarjetas índice del modelo. Las tarjetas que colaboran deben estar separadas ya que ningún revisor debe poseer dos tarjetas que colaboren.
II. Todos los escenarios de casos de uso deben organizarse en categorías.
III. El director de la revisión lee el caso de uso con atención. Cuando llega a un objeto identificado, se pasa la señal a la persona que posee la clase tarjeta índice correspondiente.
IV. Cuando se pasa la señal, la persona que posee la tarjeta de la clase describe las responsabilidades mencionadas en la tarjeta y el grupo determinará si una (o más) de las responsabilidades satisface el requisito del caso de uso.
V. Si las responsabilidades y colaboraciones mencionadas en las tarjetas índice no pueden acomodarse al caso de uso, se hacen modificaciones a las tarjetas que pueden ser la definición de nuevas clases y sus tarjetas CRC o la especificación de responsabilidades y colaboraciones nuevas o revisadas en tarjetas existentes. Este modus-operandi continúa hasta terminar el caso de uso.

El modelo CRC es una primera representación del modelo de análisis para un sistema OO. Puede ser "probado" realizando una revisión dirigida por casos de uso derivados del sistema.

Definición de Jerarquía de Clases

(Ver en PPTs tipos de estructuras (Todo parte / Herencia))

Temas / Paquetes (UML)

Modelo Objeto-Relación (MOR)

(Ver en PPTs)

Modelo Objeto-Comportamiento (MOC)

(Ver en PPTs)

Análisis del Dominio

El AOO puede ocurrir a muchos niveles diferentes de abstracción.

El nivel de abstracción más alto (el nivel de empresa) no es de nuestro interés. Al nivel de abstracción más bajo, el AOO cae dentro del alcance general de la Ingeniería de Software orientada u objetos y será el centro de nuestra atención. En realidad, nos concentraremos en el AOO que se realiza a un nivel medio de abstracción. Esta actividad, llamada análisis del dominio, tiene lugar cuando una organización desea crear una biblioteca de clases reutilizables ampliamente aplicables a una categoría completa de aplicaciones.

Análisis de reusabilidad y del dominio

Las tecnologías de objetos están influenciadas por la reusabilidad. Por ejemplo, si de dos equipos de trabajo A y B, el equipo B usa una biblioteca de clases robusta, resulta que:

- El equipo B finalizará el proyecto mucho antes que el equipo A. (**tiempo**)
- El costo del producto del equipo B será significativamente más bajo que el costo del producto del equipo A. (**costo**)
- El producto del equipo B tendrá menos defectos distribuibles que los del producto del A (**eficiencia**)

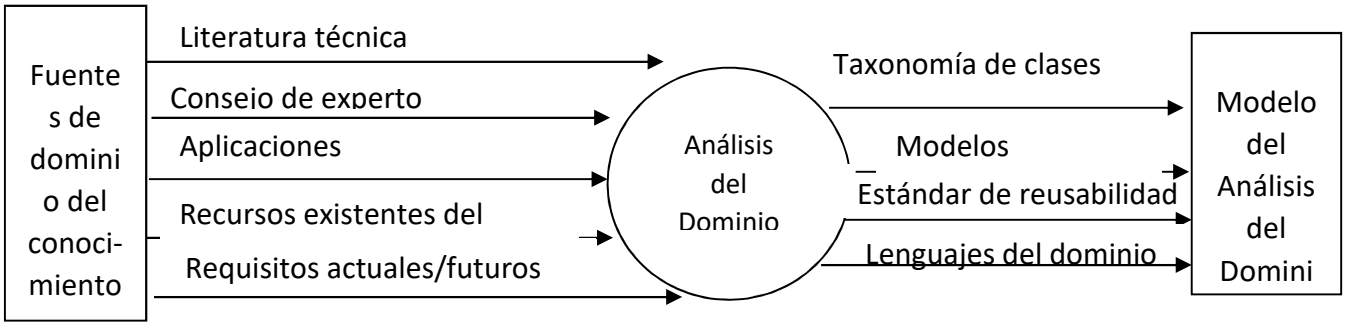
Para determinar cuáles serían las entradas adecuadas al crear esa biblioteca robusta se tuvo que aplicar el análisis del dominio.

El proceso de análisis del dominio

> El análisis del dominio del software es la identificación, análisis y especificación de requisitos comunes **de un dominio de aplicación específico,** normalmente para su reusabilidad en múltiples proyectos dentro del mismo dominio de aplicación...
> [el AOO del dominio es] la identificación, análisis y especificación de capacidades comunes y reusables dentro de un dominio de aplicación específico, en términos de objetos, clases, submontajes y marcos de trabajo comunes...

El objetivo del análisis del dominio es claro: encontrar y crear aquellas clases ampliamente aplicadas, de tal manera que sean reutilizables.

El siguiente diagrama ilustra las entradas y salidas clave para el proceso de análisis del dominio:



Se examinan las fuentes del dominio de conocimiento en un intento de identificar objetos que se puedan reusar a lo largo de todo el dominio. Esencialmente, el análisis del dominio es muy similar a la ingeniería del conocimiento, donde el ingeniero investiga un área de interés específico, intentando extraer hechos claves que se puedan usar para la construcción de un sistema experto o una red neuronal artificial. Durante el análisis del dominio ocurre la extracción de objetos (y clases).

El proceso de análisis del dominio puede caracterizarse por una serie de actividades que comienzan con la identificación del dominio a investigar y termina con la especificación de los objetos y clases que caracterizan este dominio.

Actividades del Análisis del Dominio:

1) Definir el dominio a investigar: el analista primero debe aislar el área de negocio, tipo de sistema o categoría del producto de interés.
Luego se deben extraer los "elementos" OO (especificaciones, diseños y códigos para clases de aplicaciones OO ya existentes, clases de soporte, bibliotecas de componentes comerciales ya desarrolladas (CYD) relevantes al dominio y pasos de prueba) y los elementos no OO (políticas, procedimientos, planes, estándares y guías, partes de aplicaciones no OO (incluyendo especificación, diseño e información de prueba), métricas, y CYD del software no OO).

2) Clasificar los elementos extraídos del dominio: los elementos se organizan en **categorías** y se establecen las características generales que definen la categoría. Se propone un esquema de clasificación para las categorías y se definen convenciones para la nomenclatura de cada elemento. Se establecen jerarquías de clasificación en caso de ser apropiado.

3) Recolectar una muestra representativa de aplicaciones en el dominio: para hacerlo, el analista debe asegurar que la aplicación en cuestión tiene elementos que caen dentro de las categorías ya definidas. Debido a que se observa que, durante las primeras etapas de uso de las tecnologías de objetos, una organización del software tendrá muy pocas o casi ninguna aplicación OO, el analista del dominio debe **identificar los objetos conceptuales (no físicos) en cada aplicación [no OO]**.

4) Analizar cada aplicación dentro de la muestra: las siguientes etapas ocurren durante el proceso de análisis del dominio:

- Identificar objetos contenidos reusables.

- Indicar razones que hacen que el objeto haya sido identificado como reusable.
- Definir adaptaciones al objeto que también puede ser reusable.
- Estimar el porcentaje de aplicaciones en el dominio que pueden reutilizar el objeto.
- Identificar los objetos por nombre y usar técnicas de gestión de configuración para controlarlos.

Adicionalmente, una vez que se han definido los objetos, el analista debe estimar qué porcentaje de una aplicación típica podrá construirse usando los objetos reusables.

5) Desarrollar un modelo de análisis para los objetos: el modelo de análisis servirá como base para el diseño y construcción de los objetos del dominio.

Además de estas etapas, el analista del dominio también debe crear un conjunto de líneas maestras para la reusabilidad y desarrollar un ejemplo que ilustre cómo los objetos del dominio pueden usarse para crear una aplicación nueva.

Cuando un sistema o producto del dominio es definido como estratégico a largo plazo, puede desarrollarse un esfuerzo continuado para crear una biblioteca reusable robusta con el objetivo de crear software dentro del dominio con un alto porcentaje de componentes reusables.

Las ventajas de dedicar esfuerzo a la ingeniería del dominio son: bajo costo, mayor calidad y menor tiempo de comercialización.

Modelo de Análisis Orientado a Objetos (Modelo AOO)

El proceso de AOO se adapta a los conceptos y principios básicos de análisis, aunque la terminología, la notación y actividades difieran de los usados en métodos convencionales, el AOO resuelve los mismos objetivos subyacentes.

> **El Análisis OO...** se ocupa de proyectar un modelo preciso, conciso, comprensible y correcto del mundo real.
> **El propósito del AOO** es modelar el mundo real de forma tal que sea comprensible. Para ello se debe examinar los requisitos, analizar las implicaciones que se deriven de ellos y reafirmarlas de manera rigurosa. Primero se deben abstraer características del mundo real y dejar los pequeños detalles para más tarde.

Ya que cada método de AOO posee un proceso único y una notación diferente, más aún, se adaptan a un conjunto de etapas genéricas y todos aportan una notación

que implementa un conjunto de componentes genéricas de un modelo de AOO, **Monarchi** y **Puhr** (92) definen un **conjunto de componentes de representación genéricos que aparecen en todos los modelos de AOO**.

Estos componentes se clasifican en:

Componentes estáticos: son estructurales por naturaleza, e indican características que se mantienen durante toda la vida operativa de una aplicación.

Componentes dinámicos: se centran en el control y son sensibles al tiempo y al tratamiento de eventos.
Definen como interactúa el objeto con otros a lo largo del tiempo.

Se identifican los siguientes componentes:

Vista estática de clases semánticas: se imponen los requisitos y se extraen y representan clases como parte del análisis. Estas clases persisten a través de todo el período de vida de la aplicación y se derivan en base a la semántica de requisitos del cliente.

Vista estática de los atributos: toda clase debe describirse explícitamente con atributos que aporten esta descripción y una indicación inicial de las operaciones relevantes a esta clase.

Vista estáticas de las relaciones: los objetos están "conectados" unos a otros de varias formas. El modelo de análisis debe representar las relaciones de manera tal que puedan identificarse las operaciones que afecten estas conexiones y que pueda desarrollarse un buen diseño de intercambio y que pueda desarrollarse un buen diseño de intercambio de mensajes.

Vista estática de los comportamientos: las relaciones anteriores definen un conjunto de comportamientos que se adaptan al escenario utilizado del sistema. Estos comportamientos se implementan a través de la definición de una secuencia de operaciones que los ejecutan.

Vista dinámica de la comunicación: los objetos deben comunicarse unos con otros y hacerlo basándose en una serie de mensajes que provoquen transiciones de un estado del sistema a otro.

Vista dinámica del control y manejo del tiempo: debe describirse la naturaleza y tiempo de duración de los eventos que provocan transiciones de estado.

Otra visión ligeramente diferente de las representaciones del AOO, debida a **Champeaux** y sus colegas (93), plantea que las componentes estáticas y dinámicas se identifican para el objeto internamente y para las representaciones interobjetos.

Una vista dinámica del objeto internamente quede caracterizarse por la **historia de vida del objeto**: estados que alcanza el objeto a lo largo del tiempo, al realizarse una serie de operaciones sobre sus atributos.

	Interiores del objeto	**Representaciones Interobjetos**
Componentes Estáticos	Clases Atributos Operaciones	Relaciones de objetos Estados Transacciones
Componentes Dinámicos	Historia de la vida del Objeto	Comunicación Temporización

El proceso de AOO

En realidad, el proceso de AOO no comienza con una preocupación por los objetos, sino con una comprensión de la manera en la que se usará el sistema: por las personas (si es un sistema de interacción con el hombre), por las máquinas (si el sistema está envuelto en un control de procesos, o por otros programas (si el sistema coordina y controla otras aplicaciones).

Recién una vez que se ha definido el escenario, comienza el modelado del software.

A continuación, definiremos una serie de técnicas que pueden usarse para recopilar requisitos básicos del usuario y después definen un modelo de análisis para un sistema orientado a objetos.

Diseño de Software

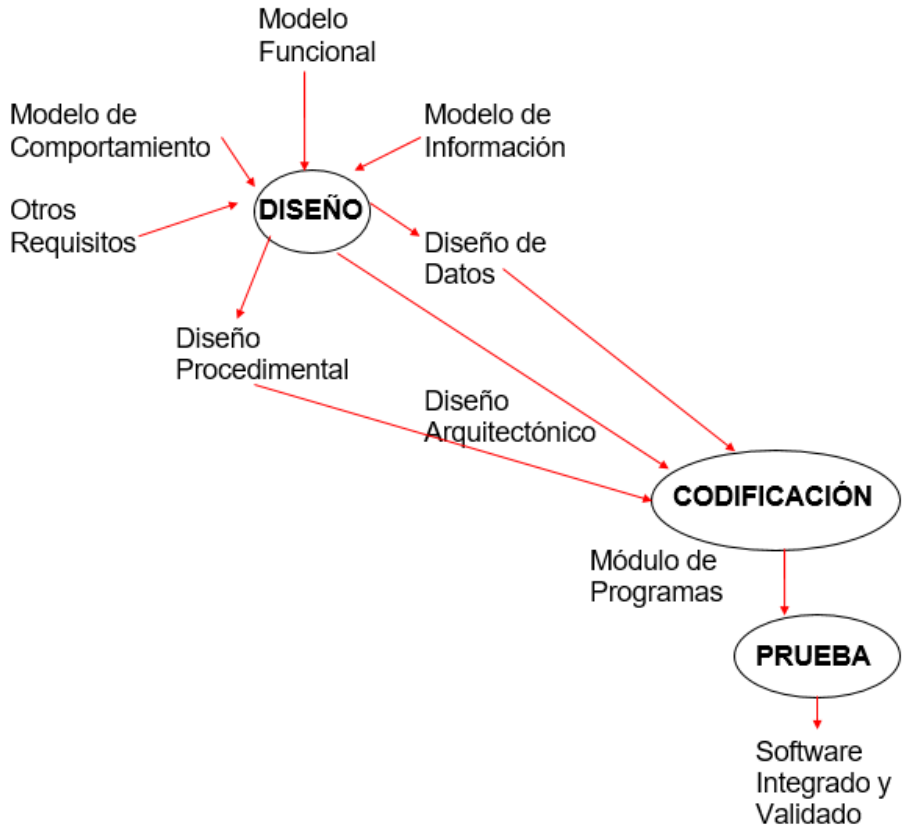
Se entiende el diseño como el primer paso de la fase de desarrollo de cualquier producto o sistema de ingeniería. Puede definirse como:

> **El proceso de aplicar distintas técnicas y principios con el propósito de definir un dispositivo, proceso o sistema con los suficientes detalles como para permitir su realización física.**

El objetivo del diseñador es producir un modelo o representación de una entidad que se construirá más adelante. El modelo que sirve de base **combina** la **intuición** y los **criterios** en base a la experiencia de construir entidades similares, un conjunto de **principios** que guían la forma en que se desarrolla el modelo, un conjunto de criterios que permiten discernir sobre la **calidad** y un **proceso de iteración** que conduce finalmente a una representación del diseño final.

El **diseño de software** se asienta en el núcleo técnico del proceso de Ingeniería de Software y se aplica independientemente del paradigma de desarrollo utilizado.

Una vez que se han establecido los requisitos del software, el diseño es la primera de tres actividades técnicas (**diseño**, **codificación** y **prueba**), cada una de las cuales transforma la información de forma tal que finalmente se obtiene el software validado.



(En el grafico notar que falta Diseño de Interfaz)

Los requisitos del programa, establecidos mediante el **Modelo de Información**, el **Modelo Funcional**, el **Modelo de Comportamiento** y los **Otros Requisitos**, alimentan el paso de **Diseño**.

Mediante alguna de las metodologías de diseño se realizan los:

- **Diseño de Datos**: Transforma el dominio de información del modelo, identificado durante el análisis, en las estructuras de datos requeridas para la implementación del software.

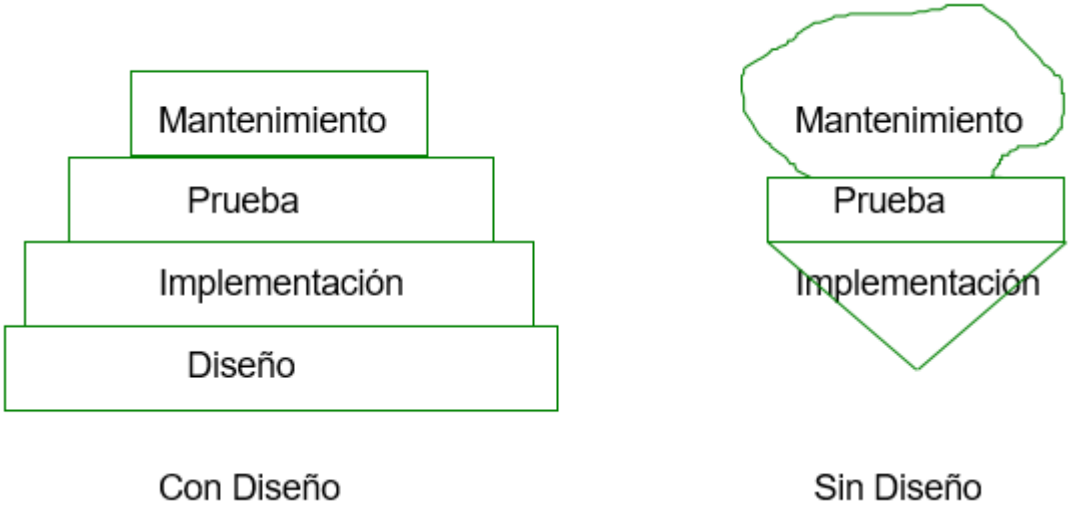
- **Diseño Arquitectónico**: Define las relaciones entre los principales elementos estructurales del programa. (Jerarquía de módulos e interfaces).

- **Diseño Procedimental**: Transforma los elementos estructurales (todos los procesos, "burbujas", al menor nivel de abstracción) en descripciones procedurales de software. A partir de él, se genera el código fuente, y para integrar y validar el software, se llevan a cabo las pruebas.

- **Diseño de Interfaz**: Especifica los componentes visuales y no visuales del sistema que interactúan con las entidades externas. Por lo general suelen ser pantallas de usuario, pero también podrían ser módulos no visuales de comunicación.

Las fases de diseño, codificación y pruebas, absorben el **75% del costo de la Ingeniería de Software**, excluyendo el mantenimiento. Acá se toman decisiones que afectarán finalmente el éxito de la implementación del programa y la facilidad de mantenimiento que tendrá el software. Estas decisiones que se toman durante el diseño hace que sea un paso fundamental del desarrollo.

El diseño es el proceso en el que se asienta la **calidad** del desarrollo de software ya que produce representaciones de software de las que puede evaluarse su calidad. **El diseño es la única forma de traducir con precisión los requisitos del cliente en un producto o sistema acabado**. El diseño sirve como base de todas las etapas posteriores de desarrollo y mantenimiento. Sin diseño nos arriesgamos a construir un sistema inestable, que falle cuando se realicen pequeños cambios, difícil de probar, cuya calidad no se pueda evaluar hasta más adelante en el proceso de Ingeniería de Software, cuando quede poco tiempo y ya se haya gastado mucho dinero.



El proceso de Diseño

> **El diseño es un proceso mediante el que se traducen los requisitos en una representación del software.**

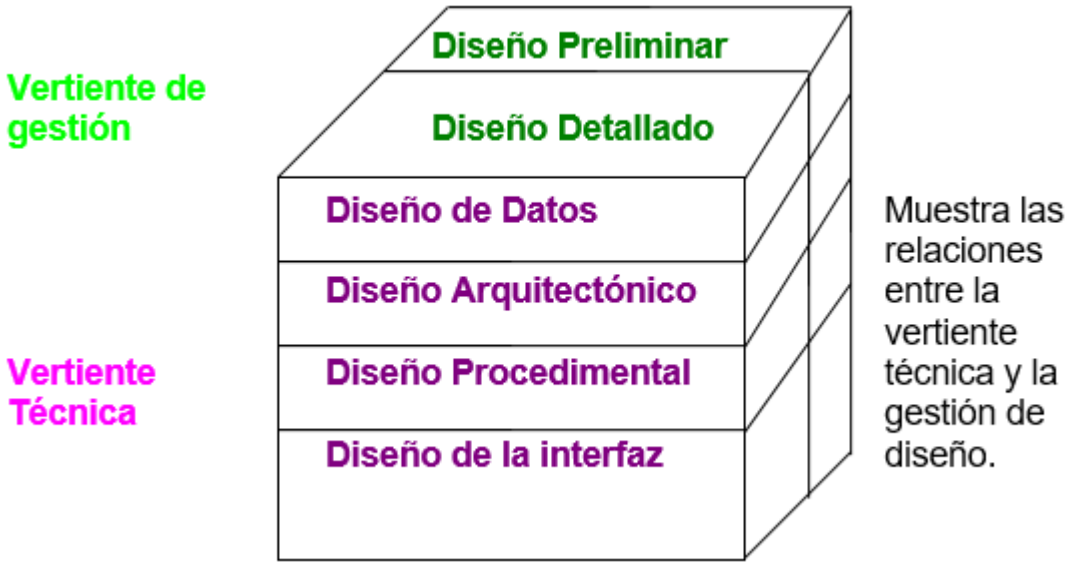
Inicialmente, la representación define una visión integral del software mientras que refinamientos posteriores conducen a una representación de diseño que se acerca mucho al código fuente. Desde el punto de vista de la gestión de proyecto, el diseño de software se realiza en **dos pasos**:

Diseño Preliminar: Se centra en la transformación de los requisitos de datos y la arquitectura del software.

Diseño Detallado: Se ocupa del refinamiento de la representación arquitectónica que lleva a una estructura de datos detallada y a las representaciones algorítmicas del software.

Dentro de estos dos pasos de diseño se llevan a cabo varias actividades de diseño diferentes: **diseño de datos**, **de arquitecturas**, **diseño procedimental** y **diseño de interfaces**.

El **diseño de la interfaz**, requerido en muchas aplicaciones modernas, establece la forma y los mecanismos para la interacción hombre-máquina.



Diseño y Calidad del Software

A lo largo del proceso de diseño, la calidad del diseño resultante se evalúa mediante una serie de revisiones **técnicas formales**, pero para evaluar la calidad de una representación de diseño se deben establecer criterios que determinen cuando es bueno.

Algunos de estos **criterios** pueden ser:

1. El diseño debe exhibir una **organización jerárquica** que haga uso inteligente del control entre los componentes del software.

2. El diseño debe ser **modular** (es decir, debe estar dividido de forma lógica en elementos que realicen funciones y subfunciones específicas).
3. El diseño debe contener **representaciones distintas** y separadas de los datos y procedimientos.
4. Un diseño debe llevar a módulos que exhiban **características funcionales** diferentes.
5. Un diseño debe llevar a **interfaces que reduzcan la complejidad** de las conexiones entre módulos y el entorno exterior.
6. Un diseño debe obtenerse mediante un **método que sea reproducible y que esté conducido por la información** obtenida durante el análisis de requisitos.

Estas **características deseables** para un buen diseño no son fáciles de conseguir, si no es mediante una aplicación de principios fundamentales de diseño, una metodología sistemática y una concienzuda revisión.

Evolución del diseño de software

La evolución del diseño de software es un proceso contínuo que se ha ido produciendo durante casi tres (3) décadas. Arranca con criterios para el desarrollo de programas modulares y métodos para mejorar la arquitectura del software de una manera descendente a principios de la década del 70. Los principios procedimentales de la definición del diseño evolucionaron hacia una filosofía denominada **programación estructurada**. Métodos posteriores prevén la traducción del flujo de datos o de la estructura de datos en una definición de diseño. Ya a fines de la década del 80 se propone un **método orientado a objetos** para la obtención del diseño.

Aunque cada metodología introduce heurísticas y notaciones propias, así como una visión particular de lo que caracteriza la calidad del diseño, todas tienen varias **características comunes**:

- Un **mecanismo para traducir** la representación del campo de la información en una representación de diseño.
- Una **notación para representar** las componentes funcionales y sus interfaces.
- **Heurísticas para el refinamiento y la partición**.

- **Criterios para la valoración de la calidad**.

Fundamentos del diseño

Existen conceptos básicos para el diseño de software que ayudan al ingeniero a responder a las siguientes preguntas:

¿Qué criterios usar para partir el software en componentes individuales?

¿Cómo se separan los detalles de una función o de la estructura de datos de la representación conceptual del software?

¿Existen criterios uniformes que definen la calidad técnica de un diseño de programas?

El comienzo de la sabiduría de un Ingeniero de Software está en reconocer la diferencia entre obtener un programa que funciones y obtener uno que funcione correctamente. Los conceptos fundamentales del diseño de software proporcionan la base necesaria para que funcione correctamente.

Estos conceptos fundamentales, que describiremos a continuación, son:

1. Abstracción
2. Refinamiento
3. Modularidad
4. Arquitectura del Software
5. Jerarquía de control
6. Estructura de Datos
7. Procedimientos del Software
8. Ocultamiento de Información

Abstracción

Cuando se considera una solución modular para cualquier problema pueden formularse muchos **niveles de abstracción**.

El **nivel superior** de abstracción establece una solución en términos amplios, usando el lenguaje del entorno del problema.

En los **niveles inferiores** de abstracción se toma una orientación más procedimental, acompañando la terminología orientada al problema con la terminología orientada a la implementación, en un esfuerzo por establecer una solución.

En el **nivel más bajo** de abstracción se establece la solución de forma que pueda implementarse directamente.

> **... la noción psicológica de "abstracción" permite concentrarse en un problema al mismo nivel de generalización, independientemente de los detalles irrelevantes de bajo nivel; el uso de la abstracción también permite trabajar con conceptos y términos que son familiares al entorno del problema, sin tener que transformarlos a una estructura no familiar... (Wasserman 83)**

Cuando se pasa de un diseño preliminar a un diseño más detallado se reduce el nivel de abstracción. El nivel de abstracción más bajo es cuando se escribe código.

Conforme nos movemos por diferentes niveles de abstracción, creamos **abstracciones de datos**, **abstracciones procedimenatales** y, a veces, **abstracciones de control**.

- Una **abstracción de datos** es una determinada colección de datos que describen un objeto (**Tipos Abstractos de Datos**).
- Una **abstracción procedural** es una determinada secuencia de instrucciones que tienen una función limitada y específica (**Procedimientos** y **Funciones**).
- Una **abstracción de control** implica un mecanismo de control de programa sin especificar los detalles internos. (**Semáforos de Sincronización** usados para coordinar las actividades de los sistemas operativos).

Refinamiento

El **refinamiento sucesivo** es una estrategia de diseño descendente propuesta por Niklaus Wirth donde cada paso de refinamiento implica algunas decisiones de diseño. La arquitectura de una programa se desarrolla en niveles sucesivos de refinamiento de los detalles procedimentales. Se desarrolla una jerarquía descomponiendo una declaración macroscópica de una función, de una forma sucesiva, hasta que se llega a las sentencias del lenguaje de programación.

Este proceso de refinamientos sucesivos es análogo al proceso de refinamiento y partición usado durante el análisis de requisitos. **La diferencia está en el nivel de detalle que se considera y no en el método**.

El refinamiento es en realidad un proceso de **elaboración**. Se comienza en un nivel superior de abstracción con una declaración de la función que no proporciona información sobre su funcionamiento ni estructura interna. Mediante el refinamiento, el diseñador amplía la declaración original dando cada vez más detalles conforme avanzan las elaboraciones.

Modularidad

Hace más de cuatro décadas que se considera el concepto de modularidad. **La arquitectura implica modularidad**: el software se divide en componentes con nombres y ubicaciones determinados, llamados **módulos**, y que se integran para satisfacer los requisitos del problema.

> **La modularidad es un atributo individual del software que permite que un programa sea intelectualmente manejable.**

El software **monolítico** (un gran programa compuesto de un único módulo) no es fácilmente abarcado por un lector, ya que el número de caminos de control, la expansión de referencias, el número de variables y la complejidad global podrían hacer imposible una correcta comprensión.

Algunas observaciones obtenidas de la resolución de problemas muestran que:

> Sea C(x) una función que define la complejidad de una problema x y sea E(x) una función que define el esfuerzo (en tiempo) requerido para resolver un problema x.
>
> Sean ahora dos problemas p_1 y p_2:
>
> **Si $C(p_1) > C(p_2) \Rightarrow E(p_1) > E(p_2)$**
>
> Este resultado es intuitivamente obvio: *Se tarda más tiempo en resolver un problema difícil.*
>
> También se ha observado que:

$$C(p_1 + p_2) > C(p_1) + C(p_2)$$

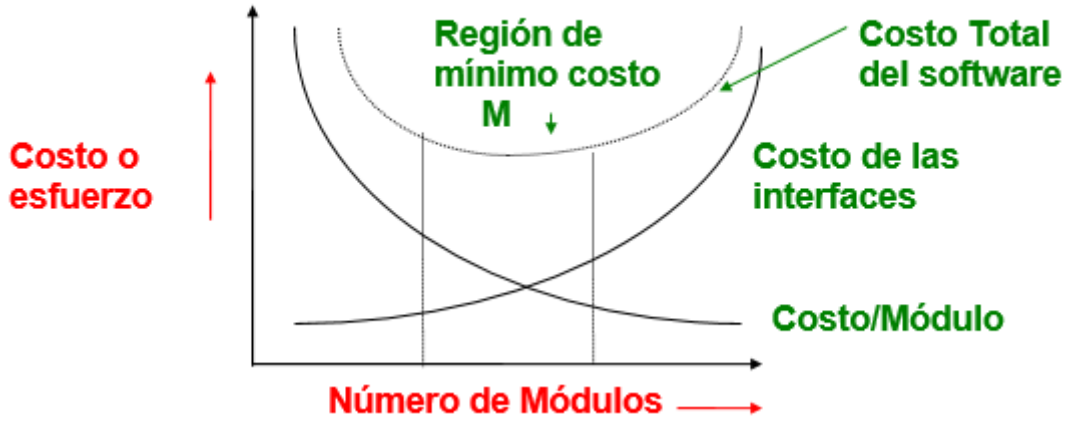
que implica que la complejidad de un problema compuesto por p_1 y p_2 es mayor que la complejidad total cuando se considera cada problema por separado.

De ambas desigualdades se deduce que:

$$E(p_1 + p_2) > E(p_1) + E(p_2)$$

Esto nos lleva a conclusión (del tipo "divide y vencerás"):

Es más fácil resolver un problema complejo cuando se divide en trozos más manejables.



El **esfuerzo o costo de desarrollo** de un módulo individual disminuye conforme aumenta el número total de módulos ya que, al haber más módulos, para un mismo conjunto de requisitos, el tamaño de cada uno es más pequeño. Pero, a medida que el número de módulos crece también crece el esfuerzo asociado a las interfaces entre módulos. La figura nos muestra que existe un número **M** de módulos para el que el costo de desarrollo es mínimo, pero no disponemos de medios eficientes para poder predecir **M** con seguridad.

En conclusión, debemos modularizar, pero siempre tratando de situarnos cerca de **M**. Más adelante veremos algunas medidas de diseño que nos ayudarán a determinar el número apropiado de módulos que debe tener el software.

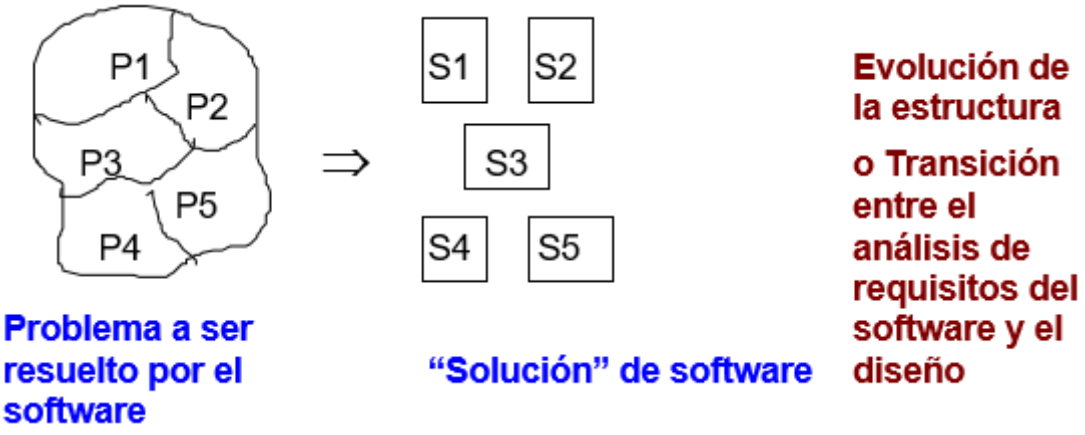
El software debe ser diseñado en forma modular, aunque su implementación posterior sea **monolítica** (por ej.: Software de tiempo real, software de microprocesadores).

Arquitectura del software

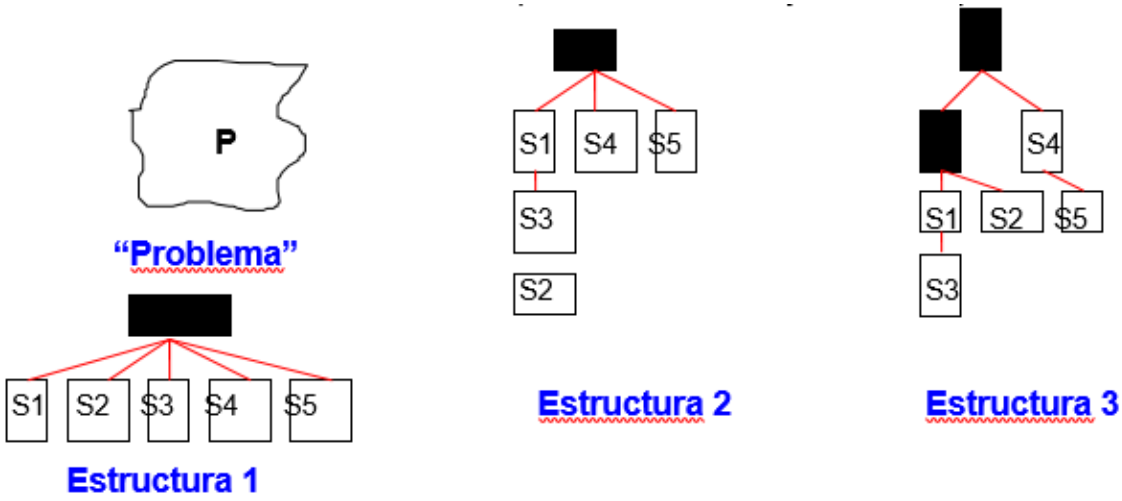
Cuando se hace referencia a la **arquitectura del software** se referencia dos características importantes del software:

- **La estructura jerárquica de los componentes procedimentales (módulos).**
- **La estructura de datos.**

La arquitectura del software se obtiene mediante el **proceso de partición**, que relaciona los elementos de una solución de software con partes de un problema del mundo real definido implícitamente durante el análisis de requisitos. **La solución aparece cuando cada parte del problema está resuelta mediante uno o más elementos de software**.



Un problema puede ser resuelto mediante diferentes estructuras ya que cada metodología de diseño que se aplique para obtener una estructura dará como resultado una estructura diferente para el mismo conjunto de requisitos del software.



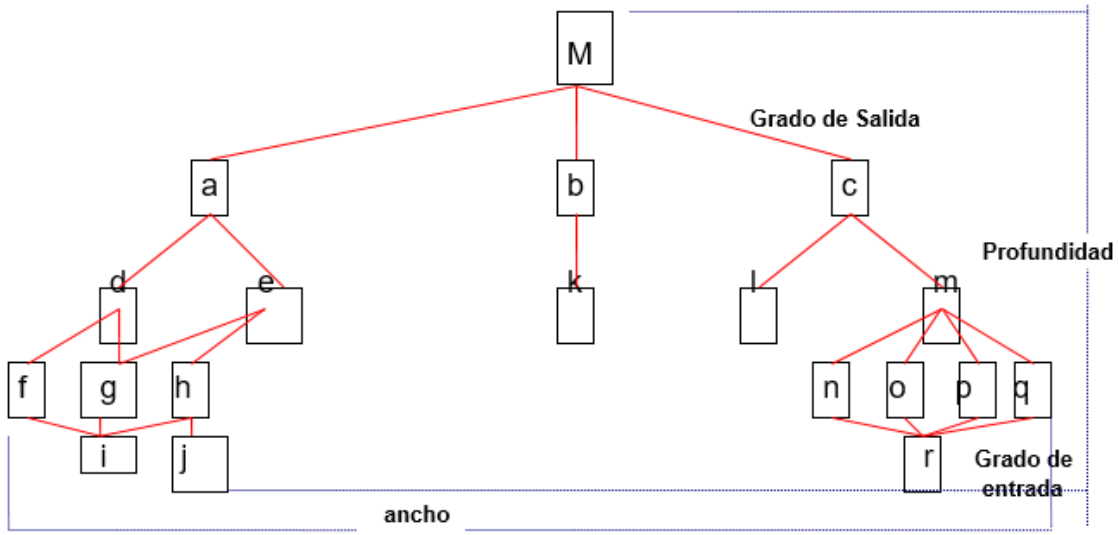
No es fácil decidir sobre cual estructura es mejor pero sí se pueden examinar algunas características de una estructura para determinar la calidad global.

Jerarquía de Control

La jerarquía de control (o estructura del programa) representa la organización (frecuentemente jerárquica) de los módulos del programa e implica una jerarquía de control.

Una jerarquía de control no representa aspectos procedimentales del software, tales como la secuencia de procesos, la ocurrencia u orden de decisiones o la repetición de operaciones.

La jerarquía de control se representa con diferentes notaciones: **Diagramas de árbol, Diagrama de Warnier-Orr y Diagramas de Jackson**. La siguiente figura muestra un diagrama de árbol que representa la estructura de un programa:



Definiciones:

Profundidad: Indicación del número de niveles de control.

Ancho: Amplitud global de control

Grado de salida: Medida del número de módulos directamente controlados por otros módulos.

Grado de entrada: Indica cuántos módulos controlan directamente a un módulo dado.

Un módulo que controla a otro se llama **superior a él**, mientras que el módulo que es controlado por otro es un **subordinado a él**. Así, M es superior a los módulos a, b y c y el módulo h es subordinado al módulo e, y en última instancia, a M.

La jerarquía de control también representa otras dos características diferentes de la arquitectura de software: la **visibilidad** y la **conectividad**.

- La **visibilidad** indica el conjunto de componentes del programa que pueden ser invocados por un componente dado o cuyos datos pueden ser usados por un componente dado, aun cuando se haga indirectamente. Por ejemplo, un módulo en un sistema OO puede tener acceso a muchos objetos de los que herede, pero puede ser que sólo use unos pocos de esos objetos. Todos los objetos son visibles para el módulo.

- La **conectividad** indica el conjunto de componentes a los que directamente se invoca o se utilizan sus datos en un determinado módulo. Por ejemplo, un módulo que directamente puede provocar la ejecución de otro módulo, está conectado a él. En un sistema de software OO, un componente puede heredar de otro componente sin una referencia explícita en el código fuente. En este caso, los componentes serán visibles, pero no estarán directamente conectados. El diagrama de estructuras indica el grado de conectividad.

Estructuras de Datos

La **estructura de datos** es una representación de la relación lógica existente entre los elementos individuales de datos. En la representación de la arquitectura del software es tan importante como la estructura del programa, debido a que **la estructura de la información afectará invariablemente el diseño procedimental final.**

La estructura de datos **define: la organización, los métodos de acceso, el grado de asociatividad y las alternativas de procesamiento** para la información.

La organización y la complejidad de una estructura de datos tan sólo está limitada por el ingenio del diseñador, aunque existe un número reducido de **estructuras de datos clásicas** que son la base para construir estructuras más sofisticadas: elementos escalares, listas enlazadas, vectores, matrices, árboles.

Procedimientos del Software

Así como la estructura del programa define la jerarquía de control, independientemente de las decisiones y secuencias de procesamiento,

El procedimiento del software se centra sobre los detalles de cada módulo individual.

El procedimiento debe proporcionar una especificación precisa del procesamiento, incluyendo la secuencia de sucesos, los puntos concretos de decisiones, la repetición de operaciones e incluso la organización/estructura de los datos.

La **relación** que existe entre la estructura y el procedimiento es que el procesamiento indicado para cada módulo debe incluir referencias a todos los módulos subordinados del módulo que se describe. En otras palabras, **la representación procedimental del software se realiza por capas**.

Ocultamiento de información

El principio de ocultamiento de información sugiere que los módulos deben especificarse y diseñarse de forma que la información (procedimientos y datos) contenida dentro de un módulo **sea inaccesible a otros módulos** que no necesiten tal información.

El ocultamiento implica que **para conseguir una modularidad efectiva hay que definir un conjunto de módulos independientes**, que se comuniquen con los otros sólo mediante la información que sea necesaria para realizar la función del software.

La abstracción ayuda a definir las entidades procedimentales que componen el software. **El ocultamiento establece y refuerza las restricciones de acceso a los detalles procedimentales internos de un módulo** y a cualquier estructura de datos localmente usada en el módulo.

Este concepto es muy útil cuando se modifica, prueba o mantiene el software porque **evita o restringe la propagación de errores**.

Diseño Modular Efectivo

La modularidad se ha convertido en un enfoque aceptado por todas las disciplinas de la ingeniería.

Un diseño modular **reduce la complejidad, facilita los cambios** (un aspecto crítico de la facilidad de mantenimiento del software) y **produce como resultado una implementación más sencilla**, permitiendo el desarrollo paralelo de las diferentes partes de un sistema.

Tipos de Módulos

La abstracción y el ocultamiento de información deben ser traducidos a características operativas de cada módulo para definir módulos en una arquitectura de software. Estas características operativas están **caracterizadas por**:

- **El Historial de incorporación**: Se refiere al momento en el que se incluye el módulo en la descripción del software en lenguaje fuente. Un subprograma es incluido mediante la generación de un código de bifurcación y enlace. Una macro de tiempo de compilación es incluida por el compilador, mediante la inserción de código, al encontrar una referencia en el código creado por el desarrollador.

- **El mecanismo de activación**: En realidad existen dos mecanismos de activación: a) Un módulo puede invocado mediante **referencia** (una sentencia de llamada) o b) En las aplicaciones en tiempo real, un módulo puede ser invocado mediante una **interrupción**, es decir, un suceso exterior produce una discontinuidad en el procesamiento que da como resultado el paso del control a otro módulo. Los mecanismos de control pueden afectar la estructura del programa.

- **El camino de control**: Describe la forma en que un módulo se ejecuta internamente. Normalmente un módulo tiene una única entrada y una única salida y ejecutan secuencialmente sus tareas. Pero, un módulo puede ser **reentrante**, es decir, se diseña de manera que no pueda modificarse a sí mismo o a las direcciones que referencie localmente y pueda de esa forma ser usado concurrentemente para más de una tarea.

Dentro de la estructura de un programa, un módulo puede **clasificarse** como:

Secuencial: Que se referencia y ejecuta sin interrupción aparente por parte del software de aplicación.

Incremental: Que puede ser interrumpido, antes de que termine, por el software de aplicación y, posteriormente, restablecida su ejecución en el punto en que se interrumpió.

Paralelo: Que se ejecuta simultáneamente con otro módulo en entornos de multiprocesadores concurrentes.

Los módulos secuenciales son los más frecuentes y están caracterizados como **subprogramas convencionales**. Los módulos incrementales, o corutinas, mantienen un puntero de entrada que permite volver a ejecutar el módulo desde el punto de interrupción. Son muy útiles en sistemas conducidos por interrupciones. Los módulos paralelos, o conrutinas, se encuentran en aplicaciones de cálculo de alta velocidad (procesamiento distribuido) que necesitan dos o más UPCs trabajando en paralelo.

Cuando se usan corutinas o conrutinas puede ser que la jerarquía de control no sea típica. Las **estructuras de control** no jerárquicas u **homólogas** requieren métodos especiales de diseño.

Independencia Funcional

Este concepto es una derivación directa de los conceptos de modularidad, abstracción y ocultamiento de información.

> **La independencia funcional se adquiere desarrollando módulos con una función "clara" y con una "aversión" a una excesiva interacción con otros módulos.**

Es decir, se trata de diseñar software de manera tal que cada módulo se centre en una sobfunción específica de los requisitos y tenga una interfaz sencilla vista desde otras partes de la estructura del software.

El software con **modularidad efectiva** o con **módulos independientes** es fácil de desarrollar (su función puede ser partida), se simplifican las interfaces, son más fáciles de probar y mantener (se limitan los efectos secundarios producidos por modificaciones en el diseño/código, se reduce la propagación de errores y se fomenta la reutilización de código).

La **independencia funcional** es la clave de un buen diseño que a su vez es la clave de la calidad del software.

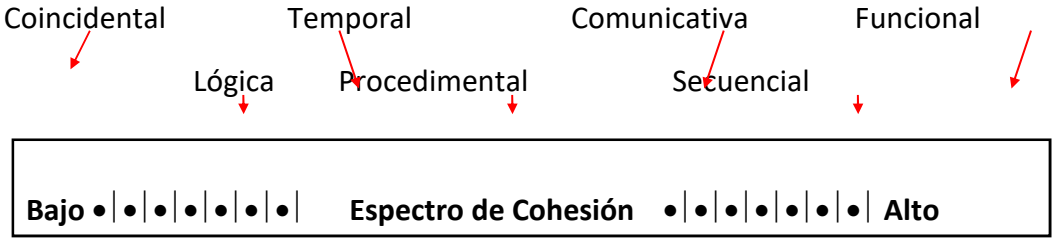
La independencia se mide con **dos criterios cualitativos**: la **cohesión** y el **acoplamiento**.

Cohesión

Es una extensión del concepto de ocultamiento de información y es una medida de la **fortaleza funcional de un módulo**.

> Un módulo cohesivo ejecuta una tarea sencilla de un procedimiento de software y requiere poca interacción con procedimientos que ejecutan otras partes de un programa. (Lleva a cabo una única tarea).

La cohesión puede presentarse como un **espectro**:



Es siempre deseable conseguir **una gran cohesión**.

- **Coincidental**: Módulo que realiza varias tareas débilmente relacionadas.
- **Lógica**: Módulo que realiza varias tareas lógicamente relacionadas.
- **Temporal**: Módulo que realiza varias tareas que se ejecutan simultáneamente.
- **Procedimental**: Los elementos de procesamiento de un módulo están relacionados y deben ejecutarse en un orden específico.
- **Comunicativa**: Todos los elementos de procesamiento se concentran sobre un área de una estructura de datos.
- **Secuencial**: Algunas partes del módulo realizan más de una tarea o existe alguna relación temporal.
- **Funcional**: Cada parte de un módulo es necesaria para realizar una única función.

Acoplamiento

Es una **medida de la interconexión** entre los módulos de una estructura de programa o bien de la interdependencia relativa entre los módulos.

El acoplamiento **depende de** la complejidad de las interfaces entre los módulos, del punto en que se hace una entrada o referencia a un módulo y de los datos que pasan a través de la interfaz.

En el diseño de software buscamos el **más bajo acoplamiento** posible. Mientras más simple sea la interconectividad entre módulos, menos propenso estará el sistema a la propagación de errores y será más fácil de comprender.

El **espectro de acoplamiento** tiene la forma:

Si acomplamiento:

directo por estampado de datos Externo de control del contenido común

Una medida de la interdependencia entre los módulos del software.

- **Sin acoplamiento directo**: Los módulos no están relacionados.
- **Acoplamiento de datos**: Existe una lista de argumentos sencilla; Se pasan datos simples; existe una correspondencia uno a uno entre elementos.
- **Acoplamiento por estampado**: Se pasa una porción de una estructura de datos (en vez de argumentos simples) a través de la interfaz del módulo.
- **Acoplamiento de control**: Se caracteriza por el paso de control entre módulos. En su forma más sencilla, el control se pasa mediante un "indicador" sobre el que se toman decisiones en un módulo subordinado o superior.
- **Acoplamiento externo**: Los módulos están ligados a un entorno externo al software. Este tipo de acoplamiento es esencial pero debe limitarse a un número pequeño de módulos. Un ejemplo es que la E/S acopla un módulo con dispositivos, formatos y

protocolos de comunicación específicos.

- **Acoplamiento común**: Acceso compartido a ítems individuales en archivos de disco o en el heap del sistema (área de datos global).
- **Acoplamiento del contenido**: Un módulo usa información de datos de control contenida dentro de los límites de otro módulo. (Bifurcación hacia la mitad del módulo).

Diseño de Datos

Es la actividad primaria de las cuatro (4) actividades de diseño realizadas durante la ingeniería de software.

El impacto de la estructura de datos sobre la estructura del programa y la complejidad procedimental hace que el diseño de datos tenga gran influencia en la calidad del software.

El proceso de diseño de datos, según Wasserman (80), puede **resumirse**:

> La actividad principal durante el diseño de datos es la selección de las representaciones lógicas de los objetos de datos (estructuras de datos), identificados durante las fases de definición y especificación de requisitos. El proceso de selección puede implicar un análisis algorítmico de las estructuras alternativas con el fin de determinar un diseño más eficiente o el uso de un conjunto de módulos (un "paquete") que proporcione las operaciones deseadas sobre alguna representación de un objeto.
>
> También es importante identificar los módulos de programa que deben operar directamente sobre las estructuras de datos lógicas. De esta manera puede restringirse el ámbito del efecto de las decisiones concretas de diseño de datos.

En todos los casos, los datos bien diseñados pueden conducir a una mejor estructura de programa, a una modularidad efectiva y a una complejidad procedimental reducida.

Considerando que el análisis de requisitos y el diseño frecuentemente se solapan, el diseño de datos puede considerarse como una parte de la tarea de análisis de requisitos.

El mismo Wasserman (80) propone el siguiente **conjunto de principios para la especificación de datos**:

1. **Los principios sistemáticos de análisis aplicados a la función y el comportamiento también deben aplicarse a los datos.**
   También se deben desarrollar y revisar las representaciones de flujo y del contenido de los datos, se deben identificar los objetos, se deben considerar organizaciones de datos alternativas y se debe evaluar el impacto de la modelización de datos sobre el diseño de software.

2. **Deben identificarse todas las estructuras de datos y las operaciones que se han de realizar sobre cada una de ellas.**
   Si se va a usar una estructura de datos formada por un conjunto de diversos elementos de datos, por ejemplo, esta estructura será manipulada por varias funciones principales. Luego de evaluar las operaciones realizadas sobre la estructura se puede definir un ADT para usarlo en el diseño posterior del software. La especificación de un ADT puede simplificar considerablemente el diseño del software.

3. **Debe establecerse y usarse un Diccionario de Datos para definir el diseño de datos y del programa.**
   Un DD representa explícitamente las relaciones entre los datos y las restricciones sobre los elementos de una estructura de datos. Es más fácil definir algorítmos que aprovechen las relaciones específicas si se dispone de un DD.

4. **Se deben posponer las decisiones de diseño de datos de bajo nivel hasta más adelante en el proceso de diseño.**
   El diseño de datos se hace mediante el proceso de refinamientos sucesivos. Primero se diseñan y evalúan los principales atributos estructurales, de forma que pueda establecerse la arquitectura de los datos. Durante el análisis de requisitos se define la organización global de los datos, se refina durante el diseño preliminar y se especifica en detalle durante el diseño detallado.

5. **La representación de una estructura de datos sólo debe ser conocida por los módulos que hagan un uso directo de los datos contenidos en la estructura.**
   Este principio alude a la importancia de los conceptos de ocultamiento de información y de acoplamiento asociado que proporcionan una valoración importante de la calidad del diseño de software, así como a la importancia de separar la visión lógica de la visión física de un objeto de datos.

6. **Se debe desarrollar una biblioteca de datos útiles y de las operaciones que se les puede aplicar.**
   Las estructuras de datos y las operaciones deben verse como un recurso para el diseño de software. Se pueden diseñar estructuras de datos para que sean reusables. Una biblioteca de **plantillas** de estructuras de datos (ADTs) puede reducir el trabajo de especificación y de diseño de los datos.

7. **El diseño de software, y el lenguaje de programación deben soportar la especificación y la realización de tipos abstractos de datos.**
   Sin esta posibilidad, puede hacerse extremadamente difícil, si no imposible, realizar una especificación directa de la estructura. (Por ejemplo, en FORTRAN que no soporta la especificación de listas enlazadas).

Diseño Arquitectónico

Su principal objetivo es **desarrollar una estructura de programa modular y representar las relaciones de control entre los módulos**.

Además, el diseño arquitectónico, mezcla las estructuras de programas y las estructuras de datos y define las interfaces que facilitan el flujo de los datos a lo largo del programa.

Diseño Procedimental

El diseño procedimental se realiza después que se ha establecido la estructura del programa y de los datos. **Idealmente**, la especificación que define los detalles algorítmicos debería explicarse en lenguaje natural, el que entienden todos los miembros del equipo de desarrollo y los cliente o usuarios.

Pero la exigencia de que esta especificación se haga sin ambigüedades ha llevado a que se deriven **formas más restringidas** para la representación de los detalles procedimentales:

- **Diagrama de Flujo**: Notación gráfica ampliamente usada para diseño procedimental, pero también la más abusada. Puede ser ineficiente, puede oscurecer el flujo de control, incrementando las posibilidades de errores y empeorando la legibilidad y el mantenimiento.
- **Diagrama de caja**
- **Diagrama de Nassi-Schneiderman**

  Diagrama N-S
- **Diagrama de Chapin**: También notaciones gráficas con 1) ámbito funcional, 2) Es imposible la transferencia arbitraria de control, 3) Se puede determinar fácilmente el ámbito de los datos locales y/o

globales y 4) Es fácil representar la recursividad.

- **Tablas de decisión**: Notación que traduce las acciones y las condiciones a una forma tabular.

- **LDP (Lenguaje de diseño de programas)**: Por todas sus características se hace la forma de diseño más usada actualmente.

UML

Introducción

UML (Unified Modeling Language) es un lenguaje que permite modelar, construir y documentar todos los elementos que componen un sistema de software. De hecho, se ha convertido en el estándar en la industria ya que fue concebido por los autores de los tres métodos más usados en la orientación a objetos: Grady Booch, Ivar Jacobson y James Rumbaugh. En la creación de UML también participaron empresas de mucho peso en la industria tales como: Microsoft, Hewlett-Packard, Oracle e IBM.

La notación se basa en los métodos de Bocch, OMT y OOSE y debido al gran prestigio de sus creadores y a que unifica las principales ventajas de cada uno de los métodos antes mencionados pone fin a la llamada "guerra de métodos" de la década de los 90.

Cabe señalar que dese su concepción, en 1994, a la fecha el lenguaje de modelado ha sufrido un cierto número de actualizaciones y estandarizaciones hasta llegar a la versión 2.0 actualmente en uso.

¿Por qué modelamos?

Ya que un modelo es una abstracción de un sistema, es decir, una simplificación de una realidad compleja,

> ***Construimos modelos para comprender mejor el sistema que se está desarrollando.***

A través delo modelado se consiguen cuatro objetivos:

a) Visualizar cómo es o cómo se desea que sea un sistema.

b) Especificar la estructura o comportamiento de un sistema.

c) Proporcionar plantillas que nos guíen en la construcción de un sistema.

d) Documentar las decisiones que se han tomado a lo largo del desarrollo del sistema.

En conclusión:

> ***Construimos modelos de sistemas complejos porque no podemos comprenderlos, en su totalidad, de una sola mirada.***

Principios del modelado

Existen, basándose en la experiencia en todas las disciplinas de ingeniería, cuatro principios básicos de modelado:

1. La elección de qué modelos crear tiene una profunda influencia sobre cómo se enfrenta un problema y cómo se da forma a la solución.

2. Todo modelo puede ser expresado a diferentes niveles de precisión.

3. Los mejores modelos están ligados a la realidad. Los modelos, cuando se apartan de la realidad, en el afán de simplificarla, no deben esconder los detalles importantes.

4. Un único modelo no es suficiente. Cualquier sistema no trivial se entiende mejor a través de un pequeño conjunto de modelos casi independientes.

Elementos de UML

El lenguaje UML es un lenguaje completo y no ambiguo que consta de tres elementos básicos: **a) Bloques de Construcción**

b) Reglas

c) Mecanismos Comunes

Todos responden al siguiente esquema gráfico:

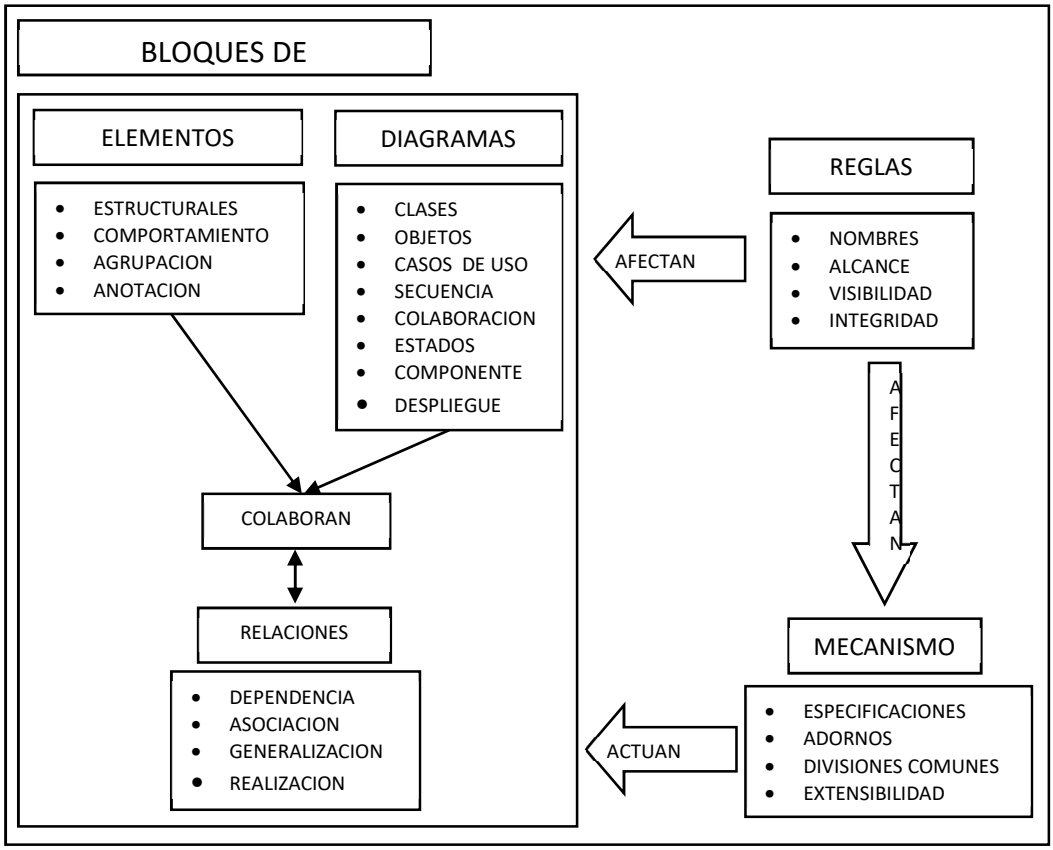


Figura II.9.1 – UML – Vista

Bloques de Construcción

El vocabulario de UML incluye tres bloques de construcción:

i) Elementos o Cosas. Abstracciones de primer nivel

ii) Relaciones: Que conectan elementos entre sí

iii) Diagramas: Agrupaciones de elementos

Elementos

Existen cuatro tipos de elementos en UML dependiendo del uso que se haga de ellos:

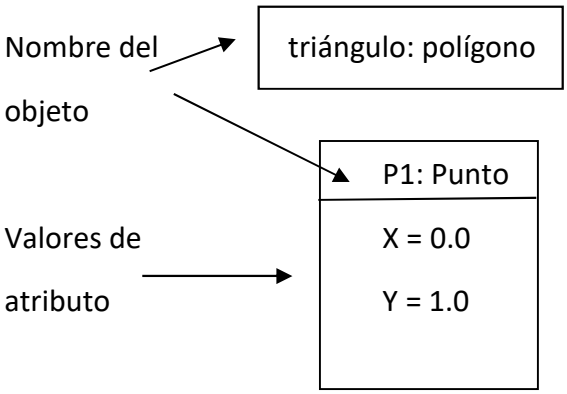
1. Elementos Estructurales
2. Elementos de Comportamiento
3. Elementos de Agrupación
4. Elementos de anotación

Elementos Estructurales

Los **elementos estructurales** son los nombres de los modelos UML. En su gran mayoría son las partes estáticas de un modelo y representan cosas que son conceptuales o materiales. Estos elementos estructurales pueden ser: Objetos, Clases, Casos de Uso, Interfaces, Componentes y Nodos.

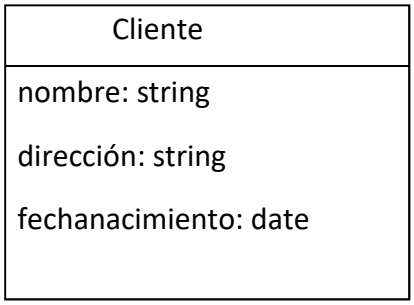
Objetos: Donde se indica su identidad, estado y comportamiento

nombre_de_objeto: nombre_de_clase

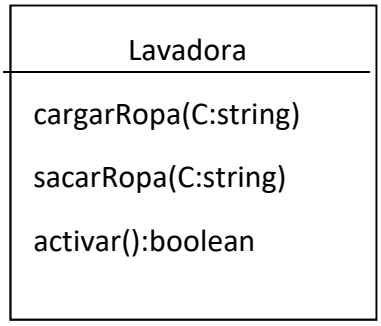


Clases: Representa un conjunto de objetos que comparten los mismos atributos, operaciones, relaciones y semántica.

Se pueden indicar la clase y sus atributos:

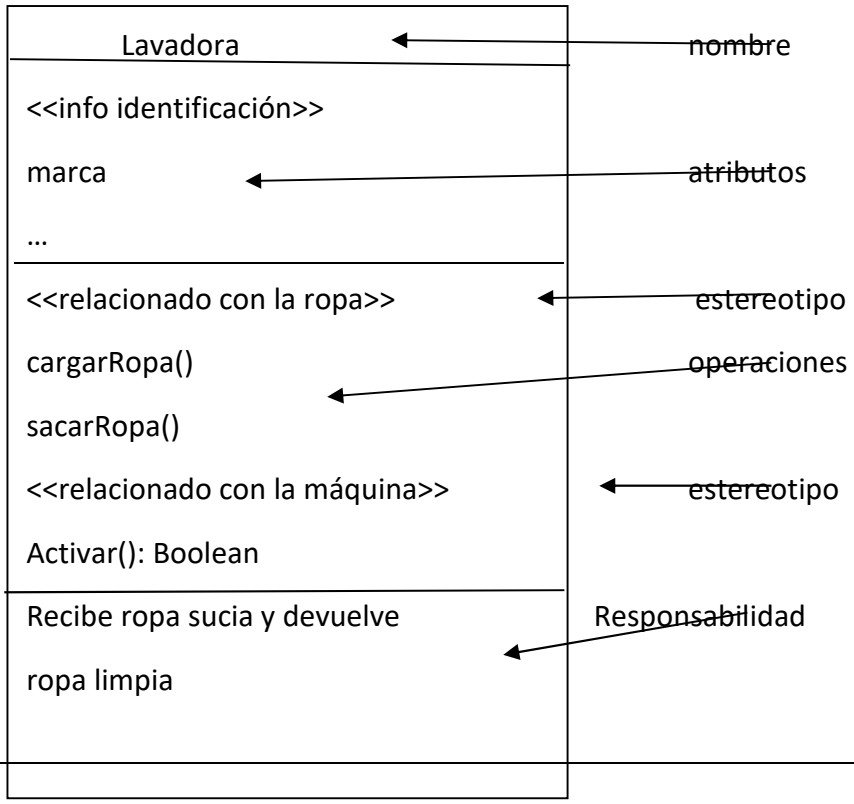


La clase y sus operaciones:

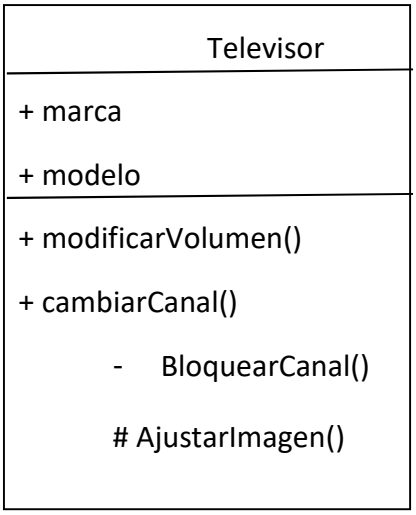


En cualquiera de los dos casos anteriores, si no se muestra en conjunto completo de atributos u operaciones se puede usar tres puntos (…) para indicar que la clase posee más atributos u operaciones de los que se muestran.

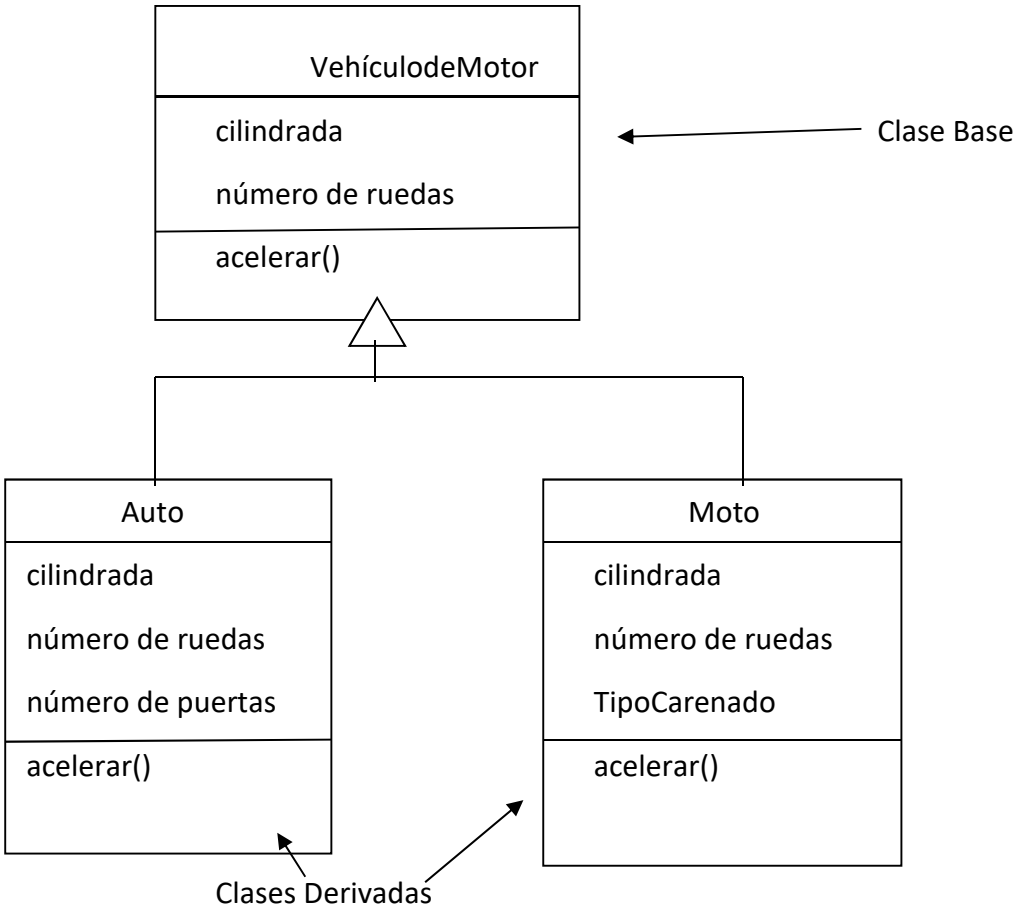
La clase, sus atributos, sus operaciones y sus responsabilidades usando estereotipos de la siguiente manera:



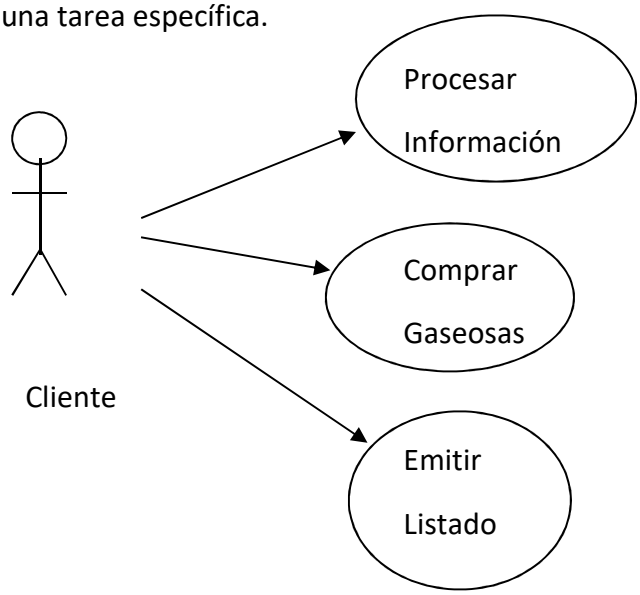
También se puede usar en la clase un marcador para indicar visibilidad de la siguiente manera: **+** (público), **-** (privado) y **#** (protegido).



También se pueden modelar los mecanismos de herencia en UML:



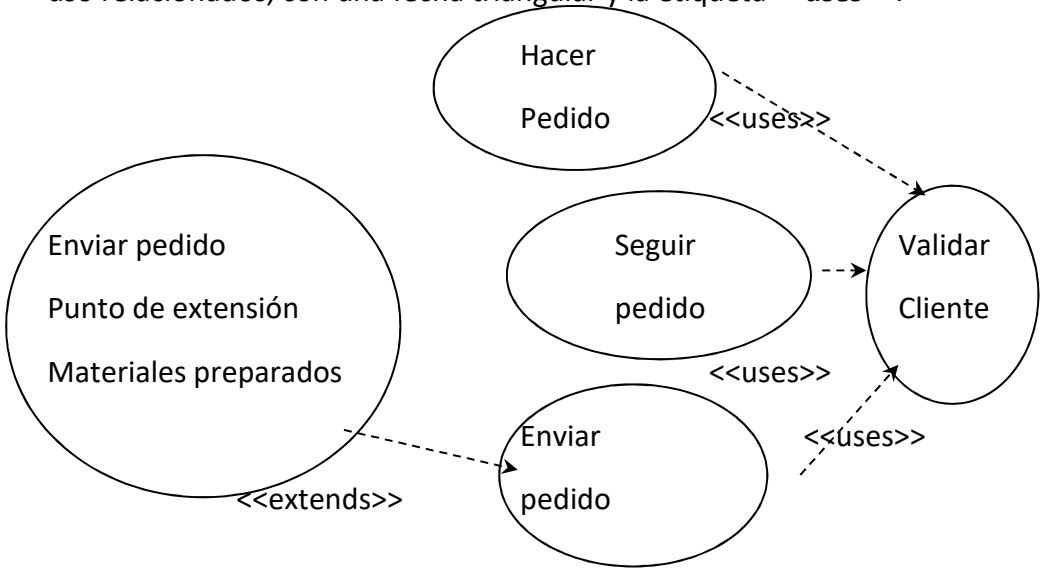
Casos de Uso: Para modelar un caso de uso con UML hay que indicar los actores participantes (representados por una figurita) y la secuencia de interacciones (representadas por una elipse) que se producen entre el actor y el sistema para llevar a cabo una tarea específica.



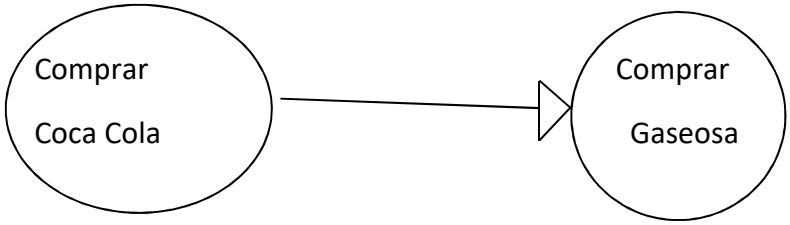
También se pueden diagramar relaciones entre casos de uso. Los siguientes son tipos de relaciones posibles entre casos de uso:

Extensión: Un caso de uso especializa a otro extendiendo su funcionalidad. La extensión sólo puede realizarse en puntos indicados de manera específica dentro de la secuencia del caso de uso base. Estos puntos se conocen como puntos de extensión. La extensión se representa con una línea que une los casos de uso , con una fecha triangular y con etiqueta ***<<extends>>***.

Inclusión: Un caso de uso utiliza los pasos de otro caso de uso dentro de sí. Un caso de uso incluido nunca parece solo, simplemente funciona como parte del caso de uso que lo incluye. Se representa con una línea que une los dos casos de uso relacionados, con una fecha triangular y la etiqueta ***<<uses>>***.



Generalización: Los casos de uso pueden heredarse entre sí. En esta herencia, el caso de uso secundario o hijo hereda las acciones y significado del primario y además agrega sus propias acciones. Puede aplicarse el caso de uso secundario en cualquier lugar en que se aplique el caso de uso primario.



Agrupamiento: Los casos de uso se pueden agrupar en paquetes de casos de uso relacionados. Los paquetes aparecen como carpetas tabulares dentro de las cuales aparecerán los casos de uso agrupados.

Interfaces: Una interfaz es una colección de operaciones que sirven para especificar el servicio que una clase o componente da al resto de las partes involucradas en un sistema. Al declarar una interfaz, se puede enunciar el comportamiento deseado de una abstracción independientemente de su implementación.

Los clientes trabajan con esa interfaz de manera independiente, así que si en un momento dado su implementación cambia, no será afectada, siempre y cuando se siga manteniendo su interfaz intacta cumpliendo las responsabilidades que tenia. A la hora de construir sistemas software es importante tener una clara separación de intereses, de forma, que cuando el sistema evolucione, los cambios en una parte no se propaguen afectando al resto. Una forma importante de lograr este grado de separación es especificar unas líneas de separación claras en el sistema, estableciendo una frontera entre aquellas partes que pueden cambiar independientemente. Al elegir las interfaces apropiadas, se pueden utilizar componentes estándar y bibliotecas para implementar dichas interfaces sin tener que construirlas uno mismo.

UML utiliza las interfaces para modelar las líneas de separación del sistema. Las interfaces no son sólo importantes para separar la especificación y la implementación de clases o componentes, sino que al pasar a sistemas más grandes, se pueden usar para especificar la vista externa de un paquete o subsistema.

La interfaz es un conjunto de operaciones que especifica algo respecto al comportamiento de una clase. Se puede pensar en una interfaz como una clase que solo tiene operaciones (sin atributos).

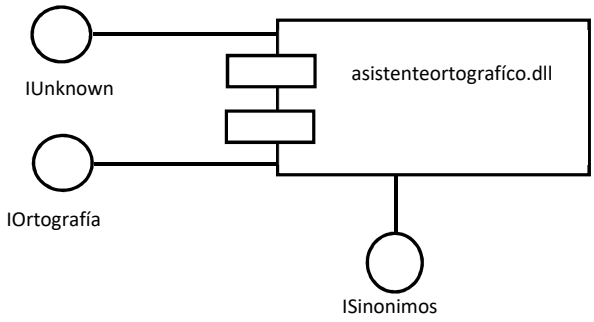


Figura II.10.11 – Interfaz – Interfaces

Una interfaz especifica un contrato para una clase o componente sin dictar su implementación. Una clase o componente puede realizar diversos interfaces, al hacer eso se compromete a cumplir esos contratos, lo que significa que proporciona un conjunto de métodos que implementan apropiadamente las operaciones definidas en el interfaz. La relación entre un componente y una interfaz se llama *Realización*

Existen dos formas de representar un interfaz en UML, la primera es mediante una circunferencia conectada a un lado de una clase o componente. Esta forma es útil cuando se quiere visualizar las líneas de separación del sistema ya que por limitaciones de estilo no se pueden representar las operaciones o las señales de la interfaz. La otra forma de representar una interfaz es mostrar una clase estereotipada que permite ver las operaciones y otras propiedades, y conectarla mediante una relación de realización con la componente o el clasificador que la contiene. Una realización se representa como una flecha de punta vacía con la línea discontinua.

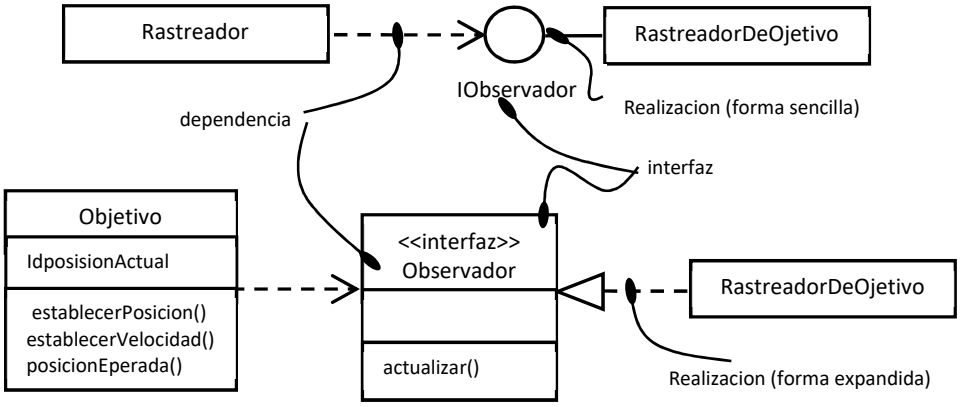


Figura II.10.12 – Interfaz – Interfaces

Componentes

Un ***componente*** es la parte física y reemplazable de un sistema, es decir, representan un bloque de construcción al modelar aspectos físicos de un sistema. Por ejemplo, una tabla (de una base de datos), un archivo de datos o ejecutable, documentos, etc. Se puede pensar en un componente como en la "personificación" en software de una clase.

En un sistema, se podrán encontrar diferentes tipos de componentes de despliegue, así como componentes que sean artefactos del proceso de desarrollo. Un componente representa típicamente el empaquetamiento físico de diferentes elementos lógicos como clases, interfaces y colaboraciones. Se representa como un rectángulo con pestañas, incluyendo solo su nombre.

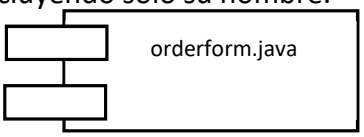


Figura II.10.13 – Componente – Componente

Una característica básica de un componente es que: "debe definir una abstracción precisa con una interfaz bien definida, y permitiendo reemplazar fácilmente los componentes más viejos con otros más nuevos y compatibles".

En UML todos los elementos físicos se modelan como componentes. Un componente representa el empaquetamiento físico de diferentes partes lógicas (clases e interfaces) que supone una parte física y reemplazable del sistema. Un componente contiene la implementación de esa parte física y reemplazable.

Existen dos formas de representar a un componente y sus interfaces:

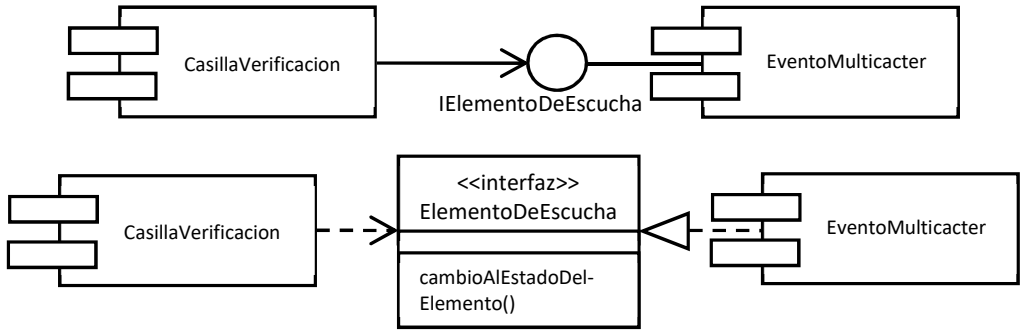


Figura II.10.14 – Comónentes e Interfaces

Nodo

Un ***nodo*** es un elemento físico que existe en tiempo de ejecución y representa un recurso computacional. Se representa como un cubo, incluyendo su nombre, también se puede agregar información, por ejemplo, los componentes que residen en el nodo.

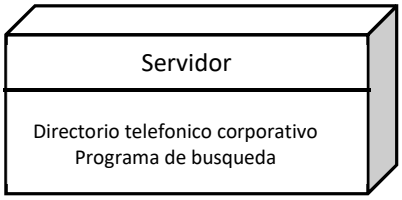


Figura II.10.15 – Nodo

Es el elemento primordial de hardware, dispone de memoria y capacidad de procesamiento (ejecuta los componentes), también existen nodos que no tienen capacidad de ejecución. Por ejemplo, computadoras (forma parte del grupo que ejecuta componentes), impresoras, monitores, modem (forman parte del segundo grupo). Un conjunto de componentes puede residir en un nodo y migrar de un modo a otro.

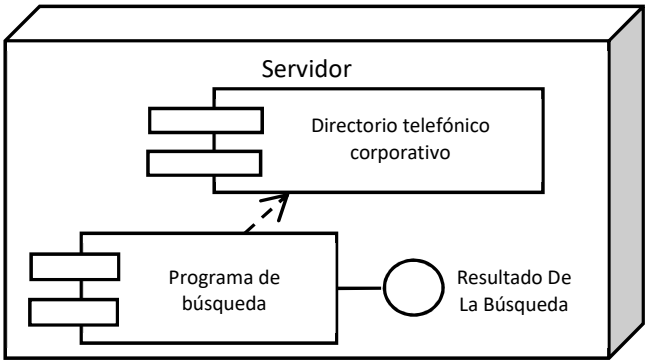


Figura II.10.16 – Nodos y sus Componentes

En muchos aspectos los nodos y los componentes tienen características similares. Diferencias y similitudes entre los componentes y los nodos.

Similitudes:

- Pueden participar en relaciones de dependencia, generalización y asociación.
- pueden anidarse
- pueden tener instancias
- pueden participar en interacciones

Diferencias:

Nodos	Componentes
Son los elementos donde se ejecutan los componentes.	Son los elementos que participan en la ejecución de un sistema.
Representan el despliegue físico de los componentes.	Representan el empaquetamiento físico de los elementos lógicos.

La relación entre un nodo y los componentes que despliega se pueden representar mediante una relación de dependencia.

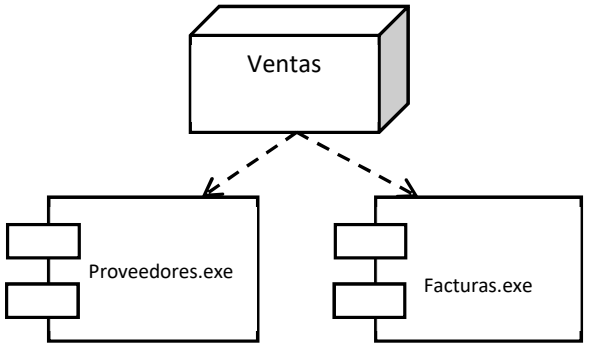


Figura II.10.17 – Relacion entre Nodo y Componente

Los tipos de relación más común entre nodos es la asociación. Una asociación entre nodos viene a representar una conexión física entre nodos.

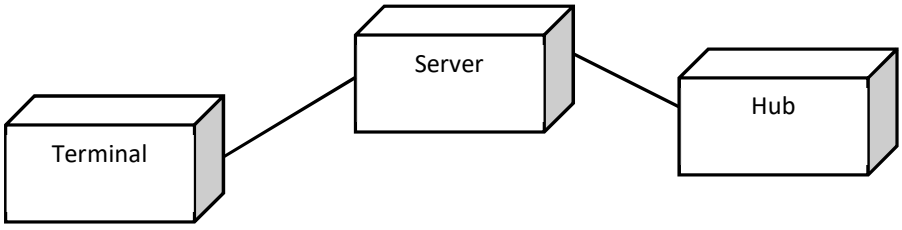


Figura II.10.18 – Relacion entre Nodos

Colaboración: define una interacción. Es una sociedad de elementos que colaboran para proporcionar un comportamiento cooperativo mayor que la suma de los comportamientos de cada elemento. Tienen dimensión tanto estructural como de comportamiento. Una clase dada puede participar en varias colaboraciones. Estas colaboraciones representan la implementación de patrones que forman un sistema. Se representa como se muestra en la figura.

Figura II.10.19 – Colaboración

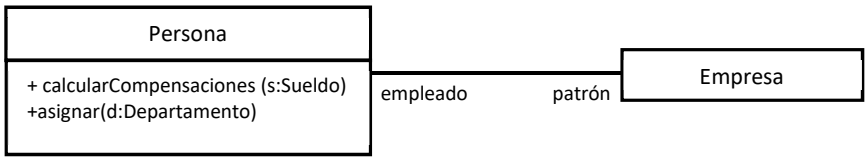
Estos son los elementos estructurales básicos más comunes que se pueden incluir en un modelo UML. También existen variaciones de estos elementos, tales como señales, utilidades (tipos de clases) y aplicaciones, documentos, archivos, bibliotecas, paginas y tablas (tipos de componentes).

XII.4.1.1.2 Elementos de Comportamiento

Los elementos de comportamiento son las partes dinámicas de los modelos UML. Estos son los verbos de un modelo, y representan comportamiento en el tiempo y el espacio.

Interacciones

Un ***enlace*** es una conexión semántica entre objetos. En general, un enlace es una instancia de una asociación, siempre que una clase tenga una asociación con otra clase, podría existir un enlace entre las instancias de dos clases; siempre que haya un enlace entre dos objetos, un objeto puede mandar un mensaje al otro.



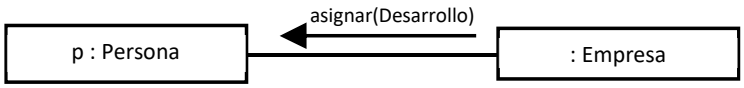


Figura II.10.20 – Interacciones – Enlaces

Un ***mensaje*** es la especificación de una comunicación entre objetos que transmite información, con la expectativa de que se desencadene alguna actividad. La recepción de una instancia de un mensaje puede considerarse la instancia de un evento.

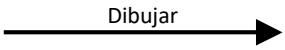


Figura II.10.21 – Interacciones – Mensaje

Un mensaje puede ser simple, sincrónico o asincrónico. Un mensaje simple es la transferencia del control de un objeto a otro. Si un objeto envía un mensaje sincrónico, esperará la respuesta a tal mensaje antes de continuar con su trabajo. Si un objeto envía un mensaje asincrónico, no esperará la respuesta a tal mensaje antes de continuar. En el diagrama de secuencias, los símbolos del mensaje varían.

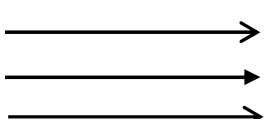


Figura II.10.22 – Interacciones – Tipos de Mensajes

Cuando se pasa un mensaje, la acción resultante es una instrucción ejecutable que constituye una abstracción de un procedimiento computacional. Una acción puede producir un cambio en el estado.

En UML se modelan varios tipos de acciones:

- *Llamada:* Invoca una operación sobre un objeto; un objeto puede enviarse un mensaje a sí mismo, lo que resulta en la invocación local de una operación.
- *Retorno:* Devuelve un valor al invocador.
- *Envío:* Envía una señal a un objeto.
- *Creación*: Crea un objeto.
- *Destrucción:* Destruye un objeto; un objeto puede "suicidarse" al destruirse a sí mismo.

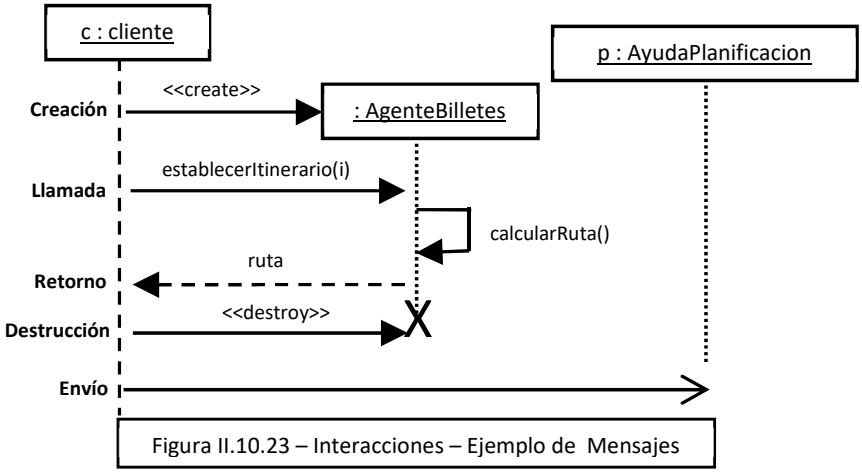


Figura II.10.23 – Interacciones – Ejemplo de Mensajes

Una ***interacción*** es un comportamiento que incluye un conjunto de mensajes intercambiados por un conjunto de objetos dentro de un contexto para lograr un propósito específico. Las interacciones se utilizan para modelar los aspectos dinámicos de las colaboraciones, que representan sociedades de objetos que juegan roles específicos, y colaboran entre sí para llevar a cabo un comportamiento mayor que la suma de los comportamientos de sus elementos.

En cualquier sistema, los objetos interactúan entre sí pasándose mensajes.

Cada iteración puede modelarse de dos formas:

- destacando la ordenación temporal de los mensajes.
- destacando la secuencia de mensajes en el contexto de una organización estructural de objetos.

Una interacción puede aparecer siempre que unos objetos estén enlazados a otros. Las interacciones aparecerán en la colaboración de objetos existentes en el contexto de un sistema o subsistema. También aparecerán interacciones en el contexto

de una operación y por último, aparecerán interacciones en el contexto de una clase. Con mucha frecuencia, aparecerán interacciones en la colaboración de objetos existentes en el contexto de un sistema o subsistema.

Maquina de estado

Una ***maquina de estado*** es un comportamiento que especifica las secuencias de estado por las que pasa un objeto a lo largo de su vida en respuesta a eventos, junto con sus relaciones a estos eventos. El comportamiento de una clase individual o colaboración de clases puede especificarse con una máquina de estados. Involucra a otros elementos, incluyendo estados, transacciones (el flujo de un estado a otro), eventos (que disparan una transacción) y actividades (la respuesta a una transacción). Un evento es una ocurrencia significativa o relevante. Un estado es la condición de un objeto en un instante de tiempo; hace referencia a los valores de sus atributos en un determinado tiempo. Una transición es una relación entre dos estados que indica cuando tiene lugar un evento; el objeto pasa de su estado al siguiente.

Un ***estado*** es una condición o situación en la vida de un objeto durante la cual satisface alguna condición, realiza alguna actividad o espera algún evento. Un objeto permanece en un estado durante un tiempo finito. Por ejemplo, un calentador (objeto) puede estar en cualquiera de los tres estados siguientes: *Apagado* (el paso de gas cerrado), *Inactivo* (el piloto esta encendido, pero no está seleccionada la temperatura deseada), *Activo* (el piloto está encendido y la temperatura está seleccionada).

Figura II.10.24 – Maquina de Estados – Estados

Un ***subestado*** es un estado animado dentro de otro. Por ejemplo, un calentador podría estar en estado Activo, pero también mientras está en estado Activo podría estar en estado de Preparación (cuando llega de la temperatura inicial a la seleccionada).

Estado ***inicial*** y ***final:*** en la máquina de estado de un objeto se pueden definir dos estados especiales. En primer lugar, el estado inicial, que indica el punto de comienzo por defecto para la máquina de estado o subestado. Un estado inicial se representa por un circulo negro relleno. En segundo lugar, el estado final, que indica que la ejecución de la máquina de estado o subestado ha finalizado. Un estado final de representa como circulo negro relleno dentro de otro circulo vacio.

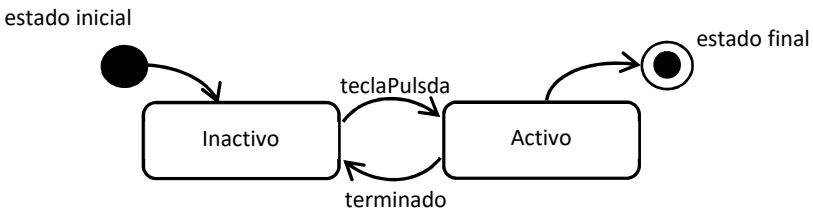


Figura II.10.25 – Estados – Estados Inicial y Final

Una transacción es una relación entre dos estados que indica que un objeto que está en un el primer estado realizará ciertas acciones y entrará en el segundo estado cuando ocurra un evento especifico y que satisfaga condiciones especificas. Cuando se produce el cambio de estado se dice que la transacción se ha disparado. Por ejemplo, un calentador podría estar en el estado Inactivo y pasar al estado Activo cuando ocurre el evento demasiadofrio (con parámetro temdeseada).

Estos son elementos básicos de comportamiento que puede incluir un modelo UML. Semánticamente estos elementos están conectados normalmente a diversos elementos estructurales, principalmente clases, colaboraciones y objetos.

XII.4.1.1.3 Elementos de Agrupación

Los elementos de agrupación son las partes organizativas de los modelos UML. Estos son cajas en las que puede descomponerse un modelo.

Paquete

Un paquete es un mecanismo de propósito general para organizar elementos en grupos. Cualquier grupo de elementos, sean estructurales o de comportamiento, puede incluirse en un paquete. Incluso pueden agruparse paquetes dentro de otro paquete. Al contrario que los componentes (que existen en tiempo de ejecución), un paquete es puramente conceptual (sólo existe en tiempo de diseño). Gráficamente se visualiza como una carpeta, incluyendo normalmente su nombre y a veces su contenido.

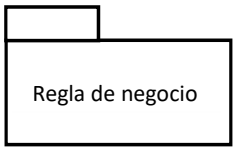


Figura II.10.26 – Paquetes

Los paquetes son elementos de agrupación básica con los cuales se puede organizar un modelo UML. También hay variaciones, tales como *frameworks*, los modelos y los subsistemas (tipos de paquetes).

La forma que tiene UML de agrupar elementos en subsistemas es a través del uso de Paquetes, pudiéndose anidar los paquetes formando jerarquías de paquetes. De hecho un sistema que no tenga necesidad de ser descompuesto en subsistemas se puede considerar como con un único paquete que lo abarca todo.

Todos los grandes sistemas se jerarquizan en niveles para facilitar su comprensión. Por ejemplo, cuando se habla de un gran edificio, se puede hablar de estructuras simples como paredes, techos y suelos, pero debido al nivel de complejidad que supone hablar de un gran edificio se utilizan abstracciones mayores como pueden ser zonas públicas, el área comercial y las oficinas. Al mismo tiempo, estas abstracciones pueden ser que se agrupen en otras mayores como la zona de alquileres y la zona de servicios del edificio, estas agrupaciones puede ser que no tengan nada que ver con la

estructura final del edificio sino se usan simplemente para organizar los planos del mismo.

La visibilidad de los elementos dentro de un paquete se puede controlar. UML proporciona una representación gráfica de los paquetes, esta notación permite visualizar grupos de elementos que se pueden manipular como un todo y en una forma permite controlar la visibilidad (elementos son visibles fuera del paquete mientras que otros permanecen ocultos) y el acceso a los elementos individuales. Los paquetes también se pueden emplear para presentar las diferentes vistas de la arquitectura de un sistema.

Cada paquete debe tener un nombre que lo distinga de otros paquetes, puede ser un nombre simple o un nombre de camino que indique el paquete donde está contenido. Al igual que las clases, un paquete, se puede dibujar adornado con valores etiquetados o con apartados adicionales para mostrar sus detalles.

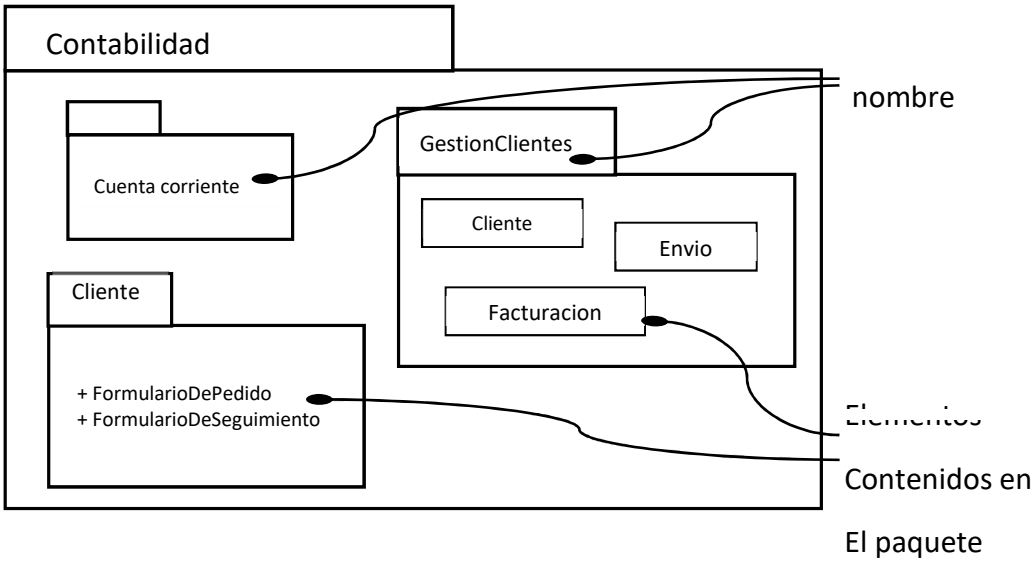


Figura II.10.27 – Paquetes – Anidación de Paquetes

Un paquete puede contener otros elementos, incluyendo clases, interfaces, componentes, nodos, colaboraciones, casos de uso, diagramas e incluso otros paquetes. Si el paquete se destruye, los elementos dentro de él también son de Cada elemento pertenece exclusivamente a un paquete.

Un paquete forma un espacio de nombres, lo que quiere decir que los elementos de la misma categoría deben tener nombres únicos en el contexto del paquete contenedor. Por ejemplo, no se pueden tener dos clases llamadas Cliente dentro de un mismo paquete, pero si se puede tener la clase Cliente dentro del paquete P1 y otra clase (diferente) llamada Cliente en el paquete P2. Las clases P1:: Cliente y P2:: Cliente son, de hecho, clases diferentes y se pueden distinguir por sus nombres de camino.

Diferentes tipos de elementos dentro del mismo paquete pueden tener el mismo nombre, por ejemplo, se puede tener dentro de un paquete la clase Temporizador y un componente llamado Temporizador. En la práctica, para evitar confusiones, es mejor asociar cada elemento a un nombre único para todas las categorías.

Los paquetes pueden contener a otros paquetes, esto significa que es posible descomponer los modelos jerárquicamente. Por ejemplo, se puede tener la clase Cámara dentro del paquete Visión y este a su vez dentro del paquete Sensores. El nombre completo de la clase será Sensores::Visión::Cámara. En la práctica es mejor evitar paquetes muy anidados, aproximadamente, dos o tres niveles de anidamiento es el límite manejable.

XII.4.1.1.4 Elementos de Anotación

Los elementos de anotación son las partes explicativas de los modelos UML. Son comentarios que se pueden aplicar para describir, clasificar y hacer observaciones sobre cualquier elemento de un modelo. Hay un tipo principal de elemento de anotación llamado nota. Una nota es simplemente un símbolo para mostrar restricciones y comentarios junto a un elemento o una colección de elementos. Gráficamente se representa como un rectángulo con una esquina doblada, junto con un comentario textual o gráfico.

Este elemento es el objeto básico de anotación que se puede incluir en un modelo UML. Típicamente, las notas se utilizan para adornar los diagramas con restricciones o comentarios que se expresan mejor en texto informal o formal. También hay variaciones sobre este elemento, tales como requisitos (que especifican algún comportamiento deseado desde la perspectiva externa del objeto).

Una nota con un comentario no tiene efecto semántico, es decir, su contenido no altera el significado del modelo al que está anexa. Por eso las notas se utilizan para especificar cosas como requisitos, observaciones, revisiones y explicaciones, además de representar restricciones.

Una nota puede contener cualquier combinación de texto y gráficos. Si la implementación lo permite, se puede incluir en una nota un URL activo, o incluso enlazar o incluir otro documento. De esta forma UML puede organizar todos los elementos que se generen o se usen durante el desarrollo.

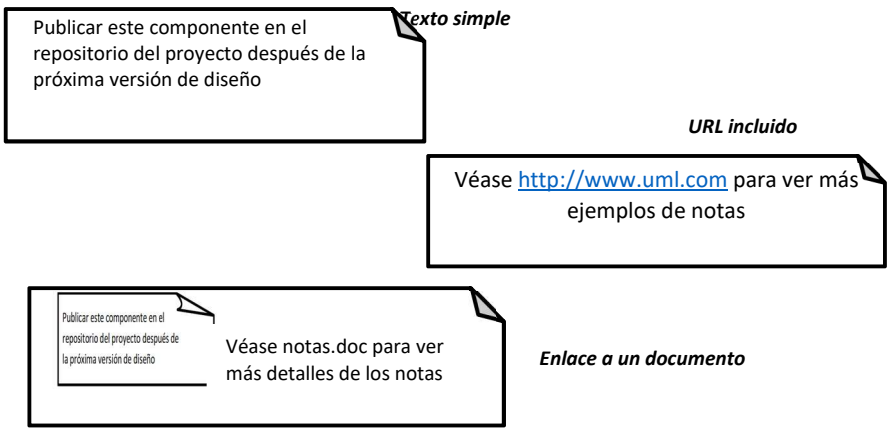


Figura II.10.28 – Ejemplo de Nota

XII.4.1.2 Relaciones

En UML la forma en que las cosas se conectan entre sí, ya sea lógica o físicamente, se modelan como relaciones. En el modelado orientado a objeto hay tres clases de relaciones muy importantes: *dependencias*, *generalizaciones* y *asociaciones*.

Las relaciones son la manera de representar las interacciones entre las clases. Tomando como ejemplo del montaje de una computadora, cada pieza interactúa con otra de una determinada manera y aunque por si solas no tienen sentido todas juntas forman una computadora, esto es lo que se denomina una relación de ***asociación***, pero además hay unas que no pueden funcionar si no están conectadas a otras como por ejemplo un teclado, el cual, sin estar conectado a una CPU es totalmente inútil, además si la CPU cambiase su conector de teclado este ya no se podría conectar a no ser que cambiase el también, esto se puede representar mediante una relación de ***dependencia***. Es más, se tiene un disco duro, un CD-ROM. Para referirse a todos estos tipos de discos se podría generalizar diciendo que se tiene una serie de discos con ciertas propiedades comunes, como pueden ser la capacidad, la tasa de transferencia en lectura y escritura, esto es lo que se denomina una relación de ***generalización***.

Estos tres tipos de relaciones cubren la mayoría de las formas importantes en que colaboran unas cosas con otras. UML proporciona una representación grafica para cada uno de estos tipos de relaciones.

Dependencia

Una ***dependencia*** es una relación semántica entre dos elementos, en la cual un cambio a un elemento (independiente) puede afectar a la semántica del otro elemento (dependiente), pero no necesariamente a la inversa. Gráficamente se representa como una línea discontinua, posiblemente dirigida.

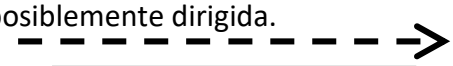


Figura II.10.29 – Dependencias

Es una relación de uso entre dos elementos de manera que un cambio en la especificación del elemento independiente puede afectar al otro elemento implicado en la relación. El elemento dependiente es aquel que necesita del elemento el independiente (otro elemento implicado en la relación) para poder cumplir sus responsabilidades. Por ejemplo suponga que se tiene una clase que representa una CPU, con sus funciones y sus propiedades. Para utilizar el método conectar() de la clase CPU, depende directamente de la clase Puerto ya que se conectará a un periférico por un puerto. A su vez la clase Teclado también depende de la clase puerto para poder transmitir los datos de entrada desde el exterior hacia la CPU.

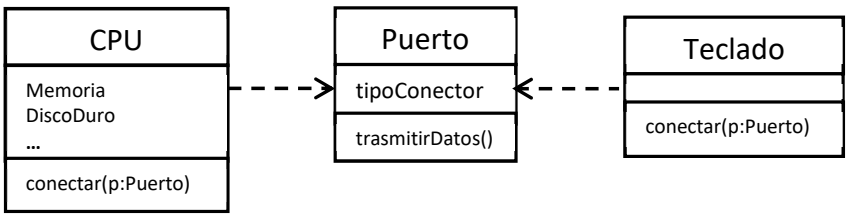


Figura II.10.30 – Relación de Dependencia

En la figura II.10.30 se observa un ejemplo de relación de dependencia. Si en algún momento la clase Puerto cambia (se modifica, por ejemplo, el atributo tipoConector de PS2 a USB) las clases CPU y Teclado (que dependen de ella) podrían verse afectadas por el cambio y dejar de funcionar. Tanto al método conectar() de la clase CPU, como el de la clase Teclado se le ha añadido más semántica representando el parámetro p de tipo Puerto. Este es un mecanismo común en UML de diseño avanzado de clases. Cuando se quiere aportar más información sobre una clase y sus métodos existe una nomenclatura bien definida para determinar más los atributos y métodos de la clase indicando si son ocultos o públicos, sus tipos de datos, parámetros que utilizan y los tipos que retornan.

Normalmente, mediante una dependencia simple sin adornos suele bastar para representar la mayoría de las relaciones de uso que aparecen, sin embargo, existen casos avanzados en los que es conveniente dotar al diagrama de más semántica.

Generalización

Una ***generalización*** es una relación de especialización/generalización en la cual los objetos especializados (el hijo) pueden sustituir a los objetos del elemento general (el padre). De esta forma, el hijo comparte la estructura y los comportamientos del padre. Gráficamente se representa como una línea continua con una punta de flecha.

Figura II.10.31 – Generalizaciones

Una generalización es una relación entre un elemento general (llamado superclase o padre) y un caso más específico de ese elemento (llamado subclase o hijo). La generalización a veces es llamada relación “es-un”, es decir, un elemento, por ejemplo, una clase Rectángulo es un elemento más general de la clase figura. La generalización implica que los objetos hijo se pueden utilizar en cualquier lugar donde aparece el padre, pero no a la inversa. La clase hijo siempre hereda todos los atributos y métodos de sus clases padre y a menudo (no siempre) el hijo extiende los atributos y operaciones del padre. Una operación de un hijo puede tener la misma signatura que en el padre pero la operación puede ser redefinida por el hijo; esto es lo que se conoce como **polimorfismo**. La generalización se representa mediante una flecha dirigida con la punta hueca. Una clase puede tener ninguno, uno o varios padres. Una clase sin padres y uno o más hijos se denomina clase raíz o **clase base**. Una clase sin hijos se denomina **clase derivada**.

Una clase con un único padre se dice que utiliza herencia simple y una clase con varios padres se dice que utiliza herencia múltiple. UML utiliza las relaciones de generalización para el modelado de clases e interfaces, pero también se utilizan para establecer generalizaciones entre otros elementos como por ejemplo los paquetes.

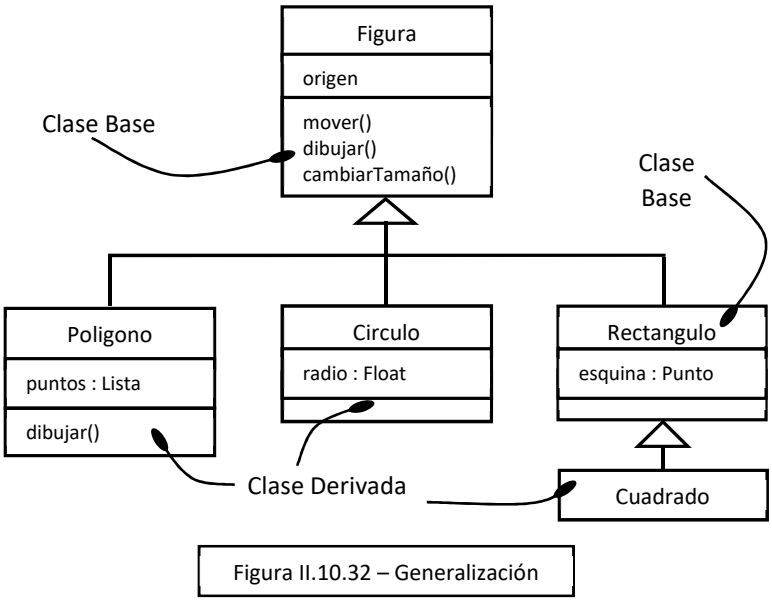


Figura II.10.32 – Generalización

Mediante las relaciones de generalización se puede ver que un Cuadrado "es-un" Rectángulo que a su vez "es-un" figura. Con esta relación se puede representar la herencia, de este modo, la clase cuadrado, simplemente por herencia se sabe que tiene dos atributos, esquina (heredado de su padre Rectángulo) y origen (heredado del padre de Rectángulo, la clase figura, que se puede decir que es su abuelo). Lo mismo ocurre con las operaciones, la clase Cuadrado dispone de las operaciones mover(), cambiarTamaño() y dibujar(), heredadas todas desde figura.

Normalmente la herencia simple es suficiente para modelar los problemas a los que nos enfrentamos pero, en ciertas ocasiones conviene modelar mediante herencia múltiple aunque vistos en esta situación se ha de ser extremadamente cuidadoso en no realizar herencia múltiple desde padres que solapen su estructura o comportamiento. La herencia múltiple se representa en UML simplemente haciendo llegar flechas (iguales que las de la generalización) de un determinado hijo a todos los padres de los que hereda. En el siguiente ejemplo se observar cómo se puede usar la herencia múltiple para representar especializaciones cuyos padres son inherentemente disjuntos pero existen hijos con propiedades de ambos. En el caso de los Vehículos, estos se pueden dividir dependiendo de por donde circulen, así se tendrá Vehículos aéreos, terrestres y acuáticos. Esta división parece que cubre completamente todas las necesidades, y así es. Dentro de los vehículos terrestres habrá especializaciones como coches, motos, bicicletas, etc. Dentro de los acuáticos se tendrá, por ejemplo, barcos, submarinos, etc. Dentro de los aéreos habrá por ejemplo, aviones, helicópteros, etc. En este momento se tiene una clasificación bastante clara de los vehículos, pero que ocurre con los vehículos que pueden circular tanto por tierra como por agua, es decir, vehículos anfibios, como por ejemplo un tanque de combate preparado para tal efecto, en este caso se puede

pensar en utilizar un mecanismo de herencia múltiple para representar que dicho tanque reúne capacidades, atributos y operaciones tanto de vehículo terrestre como acuático.

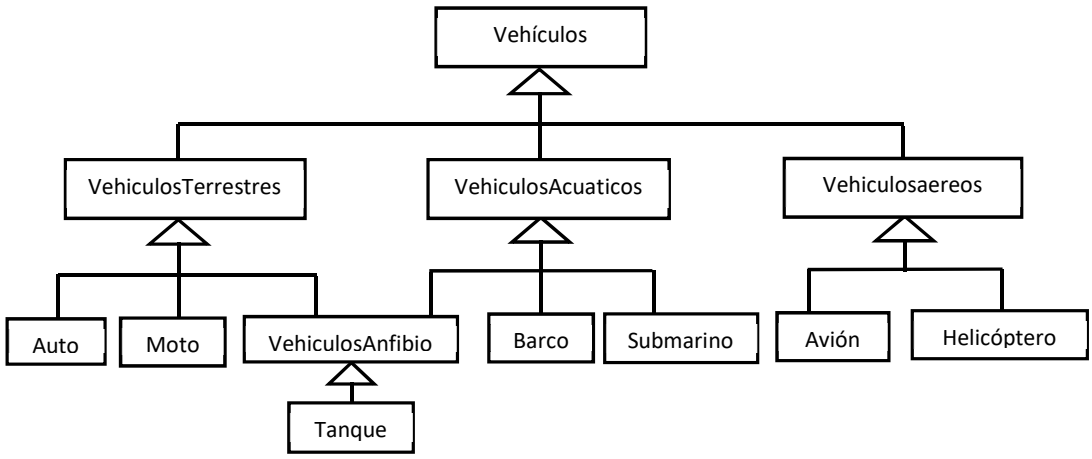


Figura II.10.33 – Herencia Múltiple

Asociación

Una ***asociación*** es una relación estructural que describe un conjunto de enlaces entre objetos. Dada una asociación entre dos clases, se puede navegar desde un objeto de una de ellas hasta uno de los objetos de la otra, y viceversa. Es posible que la asociación se dé de manera recursiva en un objeto, esto significa que dado un objeto de la clase se puede conectar con otros objetos de la misma clase. Las relaciones de asociaciones se utilizan cuando se quieren representar relaciones estructurales. Gráficamente se representa como una línea continua, incluye multiplicidad y nombres de rol.

Una asociación es una relación estructural que especifica que los objetos de un elemento están conectados con los objetos de otro elemento.

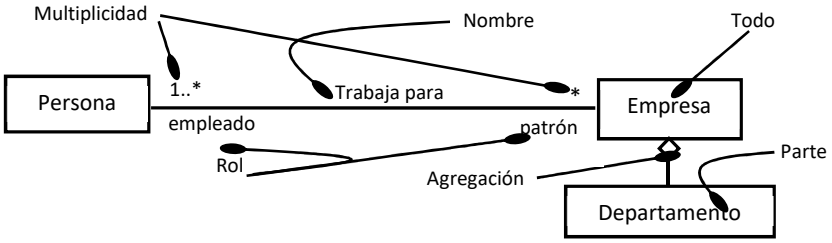


Figura II.10.34 – Asociaciones y sus partes

Las asociaciones entre dos clases se representan mediante una línea que las une. La línea puede tener elementos gráficos que expresan características particulares de la asociación. Una asociación que conecta dos clases se llama binaria. Aunque no es muy común hay asociaciones que conectan más de dos clases, éstas se llaman asociaciones n-arias.A parte de la forma básica de representar las asociaciones, mediante una línea

continua entre las clases involucradas en la relación, existen cuatro adornos que se aplican a las asociaciones para facilitar su comprensión: *nombre, rol, multiplicidad, agregación.*

El ***nombre de la asociación*** es opcional y se muestra como un texto que está próximo a la línea. Se puede añadir un pequeño triángulo negro sólido que indique la dirección en la cual leer el nombre de la asociación. En el ejemplo de la Figura II.10.35 se puede leer la asociación como "Persona trabaja para Empresa". Los nombres de las asociaciones normalmente se incluyen en los modelos para aumentar la legibilidad. Sin embargo, en ocasiones pueden hacer demasiado abundante la información que se presenta, con el consiguiente riesgo de saturación. En ese caso se puede suprimir el nombre de las asociaciones consideradas como suficientemente conocidas. En las asociaciones de tipo agregación y de herencia no se suele poner el nombre.

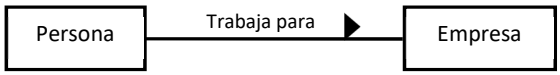


Figura II.10.35 – Asociación – Nombre

El ***rol*** se especifica para indicar papel que juega una clase en una asociación. Se representa en el extremo de la asociación junto a la clase que desempeña dicho rol. Cuando una clase participa en una asociación esta tiene un rol especifico que juega en dicha asociación. El rol es la cara que dicha clase presenta a la clase que se encuentra en el otro extremo. Las clases pueden jugar el mismo o diferentes roles en otras asociaciones.

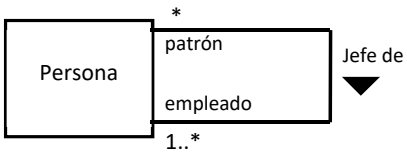


Figura II.10.36 – Asociación – Rol

La ***multiplicidad*** es una restricción que se pone a una asociación, que limita el número de instancias de una clase que pueden tener esa asociación con instancias de la otra clase. En muchas situaciones del modelado es conveniente señalar cuantos objetos se pueden conectar a través de una instancia de la asociación. Este "cuantos" se denomina multiplicidad del rol en la asociación y se expresa como un rango de valores o un valor explicito. Cuando se indica multiplicidad en un extremo de una asociación se está indicando que, para cada objeto de la clase en el extremo opuesto debe haber tantos objetos en este extremo. Puede expresarse de las siguientes formas:

- Número fijo: 1.
- Intervalo de valores: 2..5.
- Rango, con un asterisco en un extremo: 2..* significa 2 o más.

- Combinación de elementos como los anteriores separados por comas: 1, 3..5, 7,15..*.
- Con un asterisco: * . En este caso indica que puede tomar cualquier valor (cero o más).

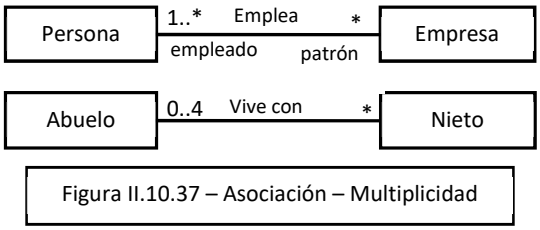


Figura II.10.37 – Asociación – Multiplicidad

Una ***agregación*** es una asociación normal entre dos clases representa una relación estructural entre iguales, es decir, ambas clases están conceptualmente al mismo nivel. A veces interesa representar relaciones del tipo "todo / parte", en las cuales una cosa representa a la cosa grande (el "todo") que consta de elementos más pequeños (las "partes"). Este tipo de relación se denomina de agregación la cual representa una relación del tipo "tiene-un".

Una agregación es sólo un tipo especial de asociación. El símbolo es un diamante colocado en el extremo en el que está la clase que representa el "todo".

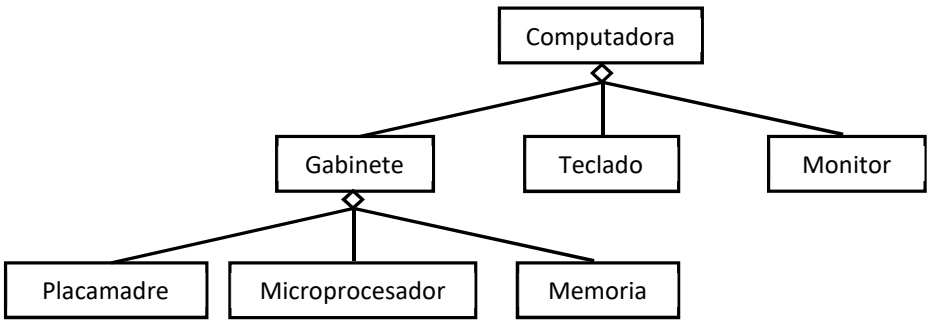


Figura II.10.38 – Asociación – Agregación

Una ***composición*** es una variación de la agregación simple que añade una semántica importante. Es una forma de agregación, con una fuerte relación de pertenencia y vidas coincidentes de la parte del todo. Las partes con una multiplicidad no fijada puede crearse después de la parte que representa el todo (la parte compuesta), una vez creadas pertenecen a ella de manera que viven y mueren con ella. Las partes pueden ser eliminadas antes que el todo sea destruido pero una vez que este se elimine todas sus partes serán destruidas. El todo, además, se encarga de toda la gestión de sus partes, creación, mantenimiento, disposición. La figura II.10.39 muestra una relación de composición donde un marco pertenece a una ventana, pero ese marco no es compartido por ninguna otra ventana, esto contrasta con la agregación simple en la que una parte puede ser compartida por varios objetos agregados. Por ejemplo una pared puede estar compart

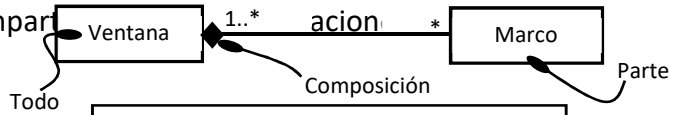


Figura II.10.39 – Asociación – Composición

La ***navegabilidad*** de una asociación se puede indicar la mediante una flecha en un extremo. Significa que es posible "navegar" desde el objeto de la clase origen hasta el objeto de la clase destino. Se trata de un concepto de diseño, que indica que un objeto de la clase origen conoce al (los) objeto(s) de la clase destino, y por tanto puede llamar a alguna de sus operaciones.La relación de ***herencia*** se representa mediante un triángulo en el extremo de la relación que corresponde a la clase más general o clase "padre". Si se tiene una relación de herencia con varias clases subordinadas, pero en un diagrama concreto no se quieren poner todas, esto se representa mediante puntos suspensivos.

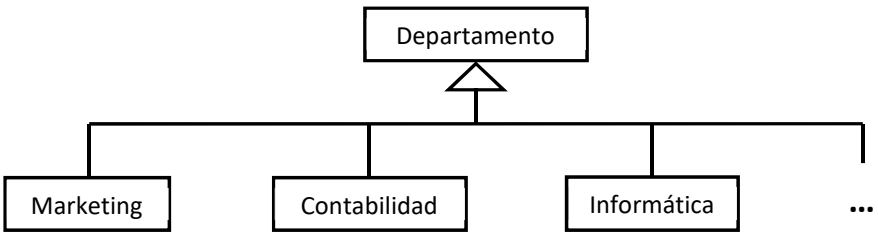


Figura II.10.40 – Herencia

Un ***elemento derivado*** es aquel cuyo valor se puede calcular a partir de otros elementos presentes en el modelo, pero que se incluye en el modelo por motivos de claridad o como decisión de diseño. Se representa con una barra "/" precediendo al nombre del elemento derivado.

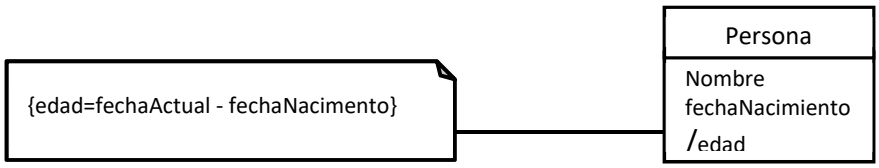


Figura II.10.41 – Elemento Derivado

Hay un cuarto tipos de relación en UML llamado *Realización*

Realización: es una relación semántica entre clasificadores, en donde un clasificador especifica un contrato que otro clasificador garantiza que se cumplirá. Se pueden encontrar relaciones de realización en dos sitios: entre interfaces y las clases o componentes que las realizan, y entre los casos de uso y las colaboraciones que los realizan.

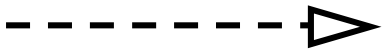


Figura II.10.42 – Realización

Estos cuatro son los elementos básicos relacionales que se pueden incluir en un modelo UML.

XII.4.1.3 Diagramas

UML está compuesto por diversos elementos gráficos que se combinan para conformar diagramas.

Un diagrama es la representación grafica de un conjunto de elementos, visualizado la mayoría de las veces como un grafo conexo de nodos (elementos) y arcos (relaciones). Un diagrama es una proyección de un sistema, es decir, una vista resumida de los elementos que constituyen un sistema.

La finalidad de los diagramas es presentar diversas perspectivas de un sistema, a las cuales se les conoce como modelos. Es importante destacar que un modelo UML describe lo que hará un sistema, pero no cómo implementarlo.

La notación UML incluye nueve tipos de diagramas:

Diagrama de clases

Los diagramas de clases facilitan las representaciones a partir de las cuales los desarrolladores podrán trabajar en lo referente al análisis. Permite al analista hablarles a los clientes en su propia terminología, lo cual hace posible que los clientes indiquen importantes detalles del sistema.

Los diagramas de clases muestran un conjunto de clases, interfaces, colaboraciones y sus relaciones. Este tipo de diagrama es el más común en los sistemas de modelado orientado a objetos. Los diagramas de clase describe la vista de diseño estática de un sistema. Los diagramas de clase que incluyen clases activas describen la vista de procesos estática de un sistema.

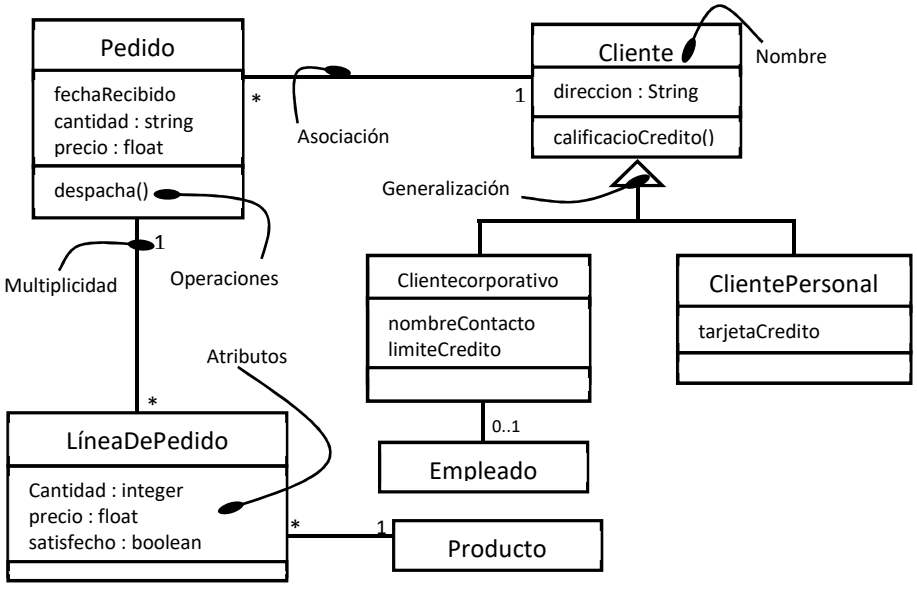


Figura II.10.41 – Diagramas – Diagrama de Clase

Diagrama de objetos

Un objeto es una instancia de una clase. UML representa a un objeto con un rectángulo, como una clase, pero el nombre de éste está subrayado. El nombre de la instancia específica se encuentra a la izquierda de los dos puntos (:), y el nombre de la clase a la derecha.

Los diagramas de objetos muestran un conjunto de objetos y sus relaciones. Estos diagramas representan "fotografías instantáneas" de instancias de los elementos que se encuentran en un diagrama de clase. Describen la vista de diseño estática o la vista de proceso estática de un sistema, al igual que los diagramas de clase, pero desde la perspectiva del mundo real.

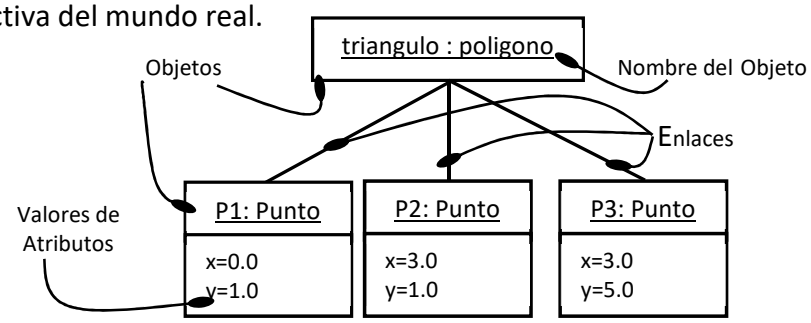


Figura II.10.44 – Diagramas – Diagrama de Objetos

Diagrama de casos de uso

Un caso de uso es una descripción de las acciones de un sistema dese el punto de vista del usuario. Para los desarrolladores del sistema, ésta es una herramienta valiosa, ya que es una técnica de aciertos y errores para obtener los requerimientos del mismo, desde el punto de vista del usuario. Esto es importante si la finalidad es crear un sistema que pueda ser utilizado por gente común y no solo por expertos en computación.

Un diagrama de casos de uso muestra un conjunto de casos de uso y actores (un tipo especial de clase) y sus relaciones. Los diagramas de casos de uso describen la vista de caso de uso estática de un sistema. Estos diagramas son importantes a la hora de organizar y modelar los comportamientos de un sistema.

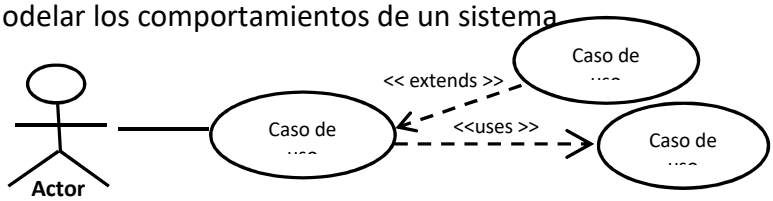


Figura II.10.45 – Diagramas – Diagrama de Casos de Uso

Diagrama de secuencias

Los diagramas de clases y los objetos representan información estática. No obstante, en un sistema funcional los objetos interactúan entre sí, y tales interacciones suceden con el tiempo. Los diagramas de secuencias de UML muestran la mecánica de la interacción con base en tiempo.

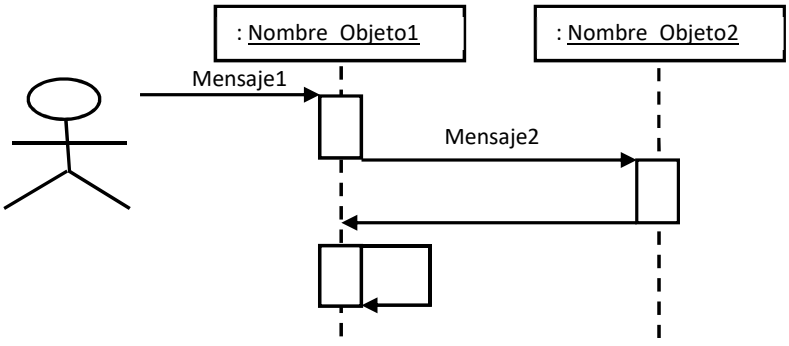


Figura II.10.46 – Diagramas – Diagrama de Secuencias

Diagrama de colaboración

Los elementos de un sistema trabajan en conjunto para cumplir con los objetivos del sistema, y un lenguaje deberá contar con una forma de representar esto. Los diagramas de colaboración fueron diseñados con ese fin.

Los diagramas de secuencias y los de colaboración son un tipo de diagrama de interacción. Un diagrama de interacción muestra una interacción, que cuenta con un conjunto de objetos y sus relaciones, incluyendo los mensajes que pueden enviarse entre ellos. Los diagramas de interacción describen la vista dinámica de un sistema. Un diagrama de secuencia resalta el orden de los mensajes en el tiempo. Un diagrama de colaboración enfatiza la organización estructural de los objetos que envían y reciben mensajes. Los diagramas de secuencia y los de colaboración son isomórficos, es decir, se pueden transformar el uno en el otro.

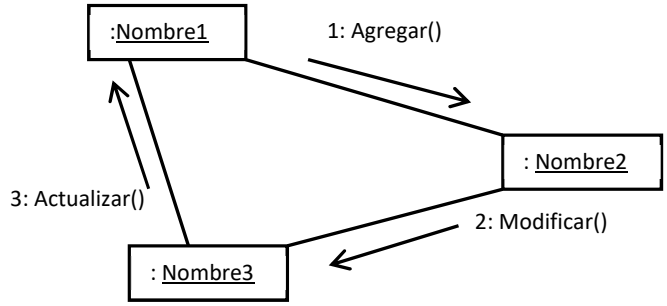


Figura II.10.47 – Diagramas – Diagrama de Colaboración

Diagrama de estados

En cualquier momento, un objeto se encuentra en un estado en particular. Una persona puede ser recién nacida, niña, adolescente, joven o adulta. El diagrama de estados de UML captura esta pequeña realidad en ese determinado instante

Un diagrama de estados muestra una máquina de estados, que consta de estados, transiciones, eventos y actividades. Describen la vista dinámica de un sistema. Estos diagramas son importantes a la hora de modelar el comportamiento de una interfaz, clase o colaboración, y enfatizan el comportamiento de un objeto ordenado por los eventos que se suceden, lo cual es especialmente útil en los sistemas de tiempo real.

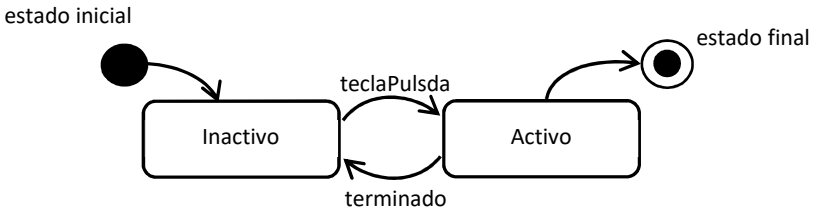


Figura II.10.48 – Diagramas – Diagrama de Estados

Diagrama de actividades

Las actividades que ocurren dentro de un caso de uso o dentro del comportamiento de un objeto, se dan normalmente en secuencias. Un diagrama de actividades es un tipo especial de diagrama de estados que muestra el flujo de actividades dentro de un sistema. Los diagramas de actividades describen la vista dinámica de un sistema. Estos diagramas son importantes a la hora de modelar el funcionamiento de un sistema y enfatizan el flujo de control entre los objetos.

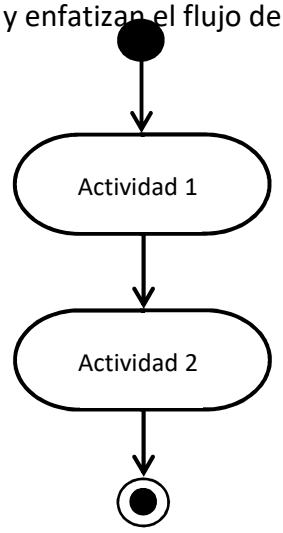


Figura II.10.49 – Diagramas – Diagrama de Actividades

Diagrama de componentes

El desarrollo moderno de software se realiza mediante componentes, lo que es particularmente importante en los procesos de desarrollo en equipo.

Un diagrama de componentes muestra las organizaciones y dependencias entre un conjunto de componentes. Los diagramas de componente describen la vista de implementación estática de un sistema. Estos diagramas se relacionan con los diagramas de clase en que un componente se corresponde, normalmente, con una o varias clases, interfaces o colaboraciones.

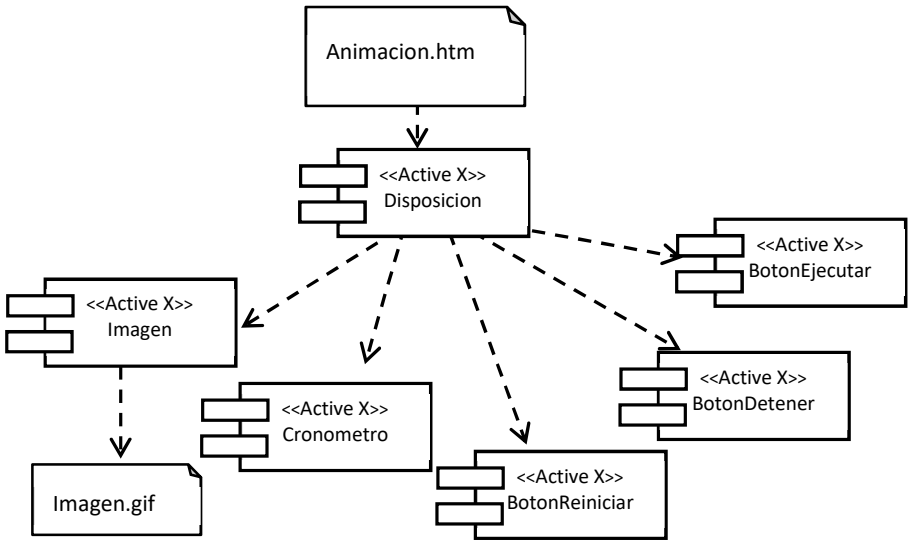


Figura II.10.50 – Diagramas – Diagrama de Componentes

Diagrama de despliegue

Un diagrama de despliegue UML muestra la arquitectura física de un sistema informático. Puede representar los equipos y dispositivos, mostrar sus interconexiones y el software que se encontrara en cada máquina. Cada computadora está representada por cubos (nodos) y las interacciones entre estas están representadas por líneas que conectan esos cubos.

Un diagrama de despliegue muestra la configuración de los nodos que se procesan en tiempo de ejecución y los componentes que están dentro de ellos. Los diagramas de despliegue describen la vista de despliegue estática de una arquitectura. Estos diagramas se relacionan con los diagramas de componente en que un nodo encierra, normalmente, uno o más componentes.

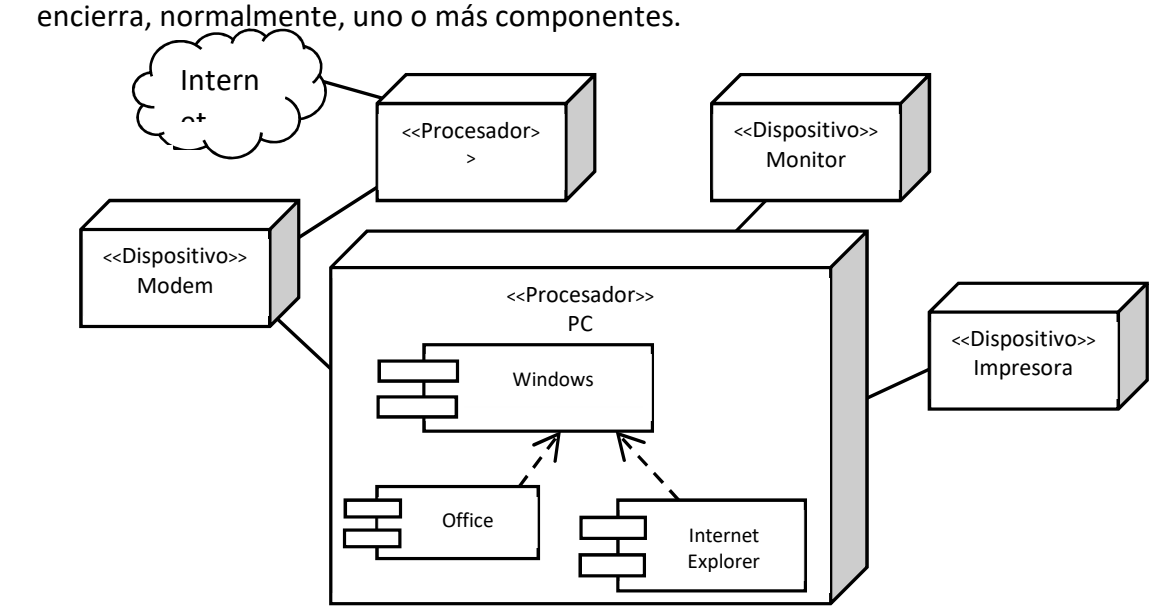


Figura II.10.51 – Diagramas – Diagrama de Despliegue

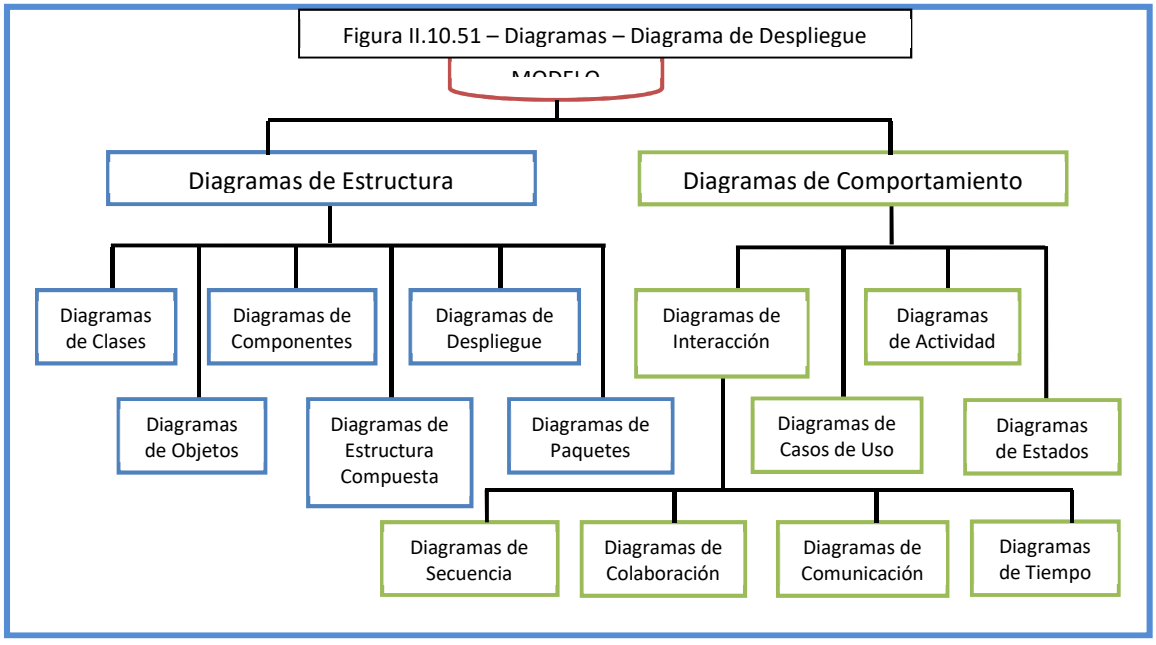


Figura II.10.52 – Diagramas de UML

Reglas

Los bloques de construcción del UML no se pueden combinar de forma aleatoria. Como cualquier otro lenguaje, UML tiene una serie de reglas que especifican a que debe parecerse un modelo bien formado. Un ***modelo bien formado*** es aquel que es semánticamente auto consistente y está en armonía con el resto de los modelos con los que se relaciona.

El UML posee reglas semánticas para:

- **Nombres**: ¿Cómo llamar a los elementos, relaciones y diagramas?
- **Ámbito**: ¿Cuál es el contexto que da un significado específico a un nombre?
- **Visibilidad**: ¿Cómo esos nombres van a ser vistos y usados por otros?
- **Integridad**: ¿Cómo los elementos se relacionan con otros apropiada y consistentemente?
- **Ejecución**: ¿Qué significa realmente ejecutar o simular un modelo dinámico?

Los modelos construidos durante el desarrollo de un sistema con gran cantidad de software tienden a evolucionar y pueden ser vistos por diferentes usuarios de forma diferente y en momentos diferentes. Por esta razón, es común en el equipo de desarrollo no solo construir modelos bien formados, sino también construir modelos que sean:

- Abreviados → Ciertos elementos se ocultan para simplificar la vista
- Incompletos → Pueden estar ausentes ciertos elementos
- Inconsistentes → No se garantiza la integridad del modelo

Estos modelos que no llegan a ser bien formados son inevitables conforme los detalles de un sistema van apareciendo y mezclándose durante el proceso de desarrollo de software. Las reglas de UML estimulan (pero no obligan) a considerar las cuestiones más importantes de análisis, diseño e implementación que llevan a tales sistemas a convertirse en bien formados con el paso del tiempo.

Mecanismos Comunes

La construcción de los bloques del UML resulta más sencilla y más armoniosa, si se realiza de acuerdo a un patrón de características comunes.

En UML se aplican de forma consistente estos cuatro mecanismos comunes:

Especificaciones

Detrás de cada parte de la notación gráfica del UML hay una especificación que proporciona una declaración textual de la sintaxis y semánticas de dicho bloque de construcción.

La notación grafica de UML se utiliza para visualizar un sistema; la especificación se usa para enunciar detalles del sistema. Hecha esta división, es posible construir modelos de forma incremental dibujando primero diagramas y añadiendo después semánticas a las especificaciones del modelo, o bien directamente, mediante la creación de una especificación primero, quizás haciendo ingeniería inversa sobre un sistema y después creando los diagramas que se obtienen como proyección de esas especificaciones.

Las especificaciones de UML proporcionan una base semántica que incluye todos los elementos de todos los modelos de un sistema y cada elemento está relacionado con otro de manera bien consistente. Los diagramas de UML son simples proyecciones visuales de esa base y cada diagrama revela un aspecto específico interesante del sistema.

Adornos

La mayoría de los elementos en el UML tienen una notación gráfica y directa que proporciona una representación visual de los aspectos más importantes del elemento. La notación de clases expone los aspectos más importantes de una clase, su nombre, sus atributos y sus operaciones. La especificación de clase puede incluir otros detalles, tales como si es abstracta o la visibilidad de sus atributos y de sus operaciones.

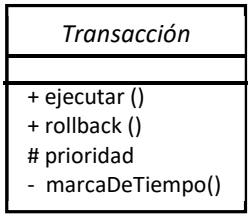


Figura II.12.1 – Adornos

Divisiones comunes

A la hora de modelar sistemas orientados a objetos, el mundo aparece dividido como mínimo en un par de formas.

En primer lugar, está la división de clase y de objeto. Una clase es una abstracción, mientras que un objeto es una manifestación concreta de dicha abstracción. En UML se puede modelar tanto las clases como los objetos.

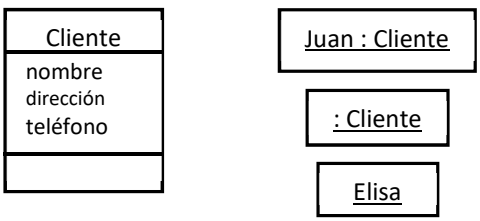


Figura II.12.2 – Divisiones Comunes – Clases y Objetos

En estas figuras hay una clase llamada Cliente, junto a tres objetos: Juan (del que se indica explícitamente que es un objeto Cliente); :Cliente (un objeto Cliente anónimo) y Elisa (el cual se ha etiquetado en su especificación como un objeto de la clase Cliente, aunque aquí no se muestra de forma explícita).

Casi todos los bloques de construcción de UML presentan este mismo tipo de dicotomía clase/objeto. Por ejemplo se puede tener casos de uso e instancias de casos de uso, componentes e instancias de componentes, nodos e instancias de nodos, etcétera. Gráficamente, UML distingue un objeto utilizando el mismo símbolo de la clase y subrayando el nombre del objeto.

Se tiene también, la separación entre interfaz e implementación. Una interfaz declara un contrato y una implementación representa una realización de ese contrato, responsable de hacer efectiva de forma fidedigna la semántica completa de la interfaz.

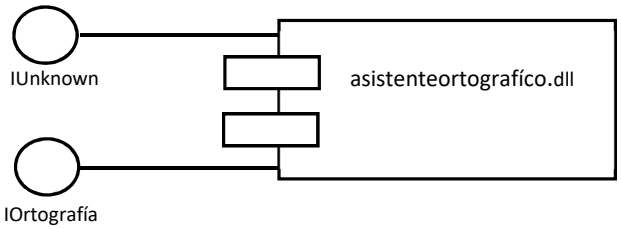


Figura II.12.3 – Interfaces e Implementaciones

En esta figura hay un componente llamado asistenteortografico.dll que implementa dos interfaces, IUnknown e IOrtografia.

Casi todos los bloques de construcción de UML presentan esta mismo tipo de dicotomía interfaz/implementación. Por ejemplo, se pueden tener casos de uso y las colaboraciones que lo realizan, así como operaciones y los métodos que lo implementan.

Mecanismos de extensibilidad

El UML proporciona un lenguaje estándar para escribir modelos de software, pero no es posible para un lenguaje cerrado expresar todos los detalles de todos los modelos en todos los dominios a lo largo de todo el tiempo. Por esta razón, el UML es un lenguaje abierto-cerrado, que se puede extender de forma controlada. Los mecanismos de extensión del UML incluyen:

Un ***estereotipo*** extiende el vocabulario del UML, permitiendo que se pueda crear nuevos tipos de bloques de construcción que son derivados a partir de los ya existentes y que son específicos a nuestro problema. Un estereotipo no es lo mismo que una clase padre en una relación de generalización padre/hijo.

Cuando se aplica un estereotipo sobre un elemento como un nodo o una clase se está extendiendo, en efecto, el vocabulario del UML, creando un nuevo bloque de construcción como cualquiera de los existentes, pero con sus propias características

(cada estereotipo puede proporcionar su propio conjunto de valores etiquetados), semántica (cada estereotipo puede proporcionar sus propias restricciones) y notación especiales (cada estereotipo puede proporcionar su propio icono).

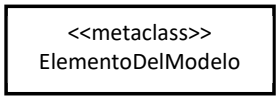


Figura II.12.4 – Mecanismo de Extensibilidad – Estereotipo

En su forma más sencilla, un estereotipo se representa como un nombre entre comillas tipográficas (por ejemplo, «nombre») y se coloca sobre el nombre del elemento. Como señal visual, se puede definir un icono para el estereotipo y mostrar ese icono a la derecha del nombre (si se utiliza notación básica para el elemento) o utilizar ese icono como símbolo básico para el elemento estereotipado.

Los ***Valores etiquetados (tagged values)***, extienden las propiedades de un bloque de construcción del UML, permitiendo agregar nueva información en la especificación de dicho elemento. Por ejemplo, si se está trabajando en un producto que atraviesa muchas versiones a lo largo del tiempo, a menudo se querrá registrar la versión y el autor de ciertas abstracciones críticas. La versión y el autor no son primitivos de UML.

En su forma más simple un valor etiquetado se ve como una cadena de caracteres entre llaves que se coloca debajo del nombre de otro elemento. Pueden ser añadidos en cualquier bloque de construcción, como una clase, introduciendo nuevos valores etiquetados.

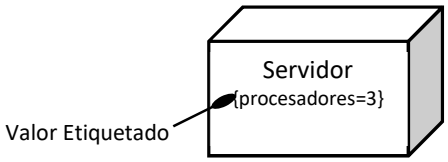


Figura II.12.5 – Mecanismo de Extensibilidad – Valores Etiquetados

Una ***restricción*** extiende las semánticas de un bloque de construcción del UML, permitiendo que se pueda añadir nuevas reglas o modificar las ya existentes. Una restricción se presenta como una cadena de caracteres entre llaves junto al elemento asociado. Por ejemplo, quizás se desee restringir la clase ColaEventos para que todas las adiciones se hicieran en orden. Se puede añadir una restricción que indique explícitamente esto para la operación

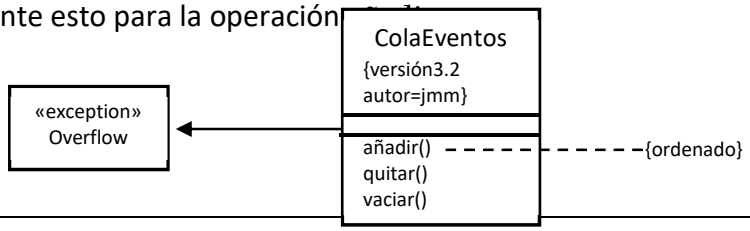


Figura II.12.6 – Mecanismo de Extensibilidad – Restricciones

En conjunto estos tres mecanismos de extensibilidad permiten configurar y extender UML para las necesidades del proyecto. Estos mecanismos también permiten a UML adaptarse a nuevas tecnologías de software, como la probable aparición de lenguajes de programación distribuidas más potentes. Se pueden añadir nuevos bloques de construcción, modificar las especificación de los existentes e incluso cambiar su semántica. Naturalmente, es importante hacer esta de forma controlada, para que a través de estas extensiones se siga siendo fiel al propósito de UML, la comunicación de información.