TECNICATURA UNIVERSITARIA EN PROGRAMACIÓN UTN-FRC

TUP

UTN

Facultad Regional Córdoba

TECNICATURA UNIVERSITARIA EN PROGRAMACIÓN

METODOLOGÍA DE SISTEMAS I

Unidad Temática 3: U.M.L. – P.U.D.

Material de Estudio

2do Año - 4to Cuatrimestre

2019

V.0.1

Índice

U.M.L Lenguaje de modelado unificado y P.U.D Proceso Unificado de Desarrollo

El lenguaje UML comenzó a gestarse en octubre de 1994, cuando Rumbaugh se unió a la compañía Rational fundada por Booch (dos reputados investigadores en el área de metodología del software). El objetivo de ambos era unificar dos métodos que habían desarrollado: el método Booch y el OMT (Object Mode-lling Tool).

El primer borrador apareció en octubre de 1995. En esa misma época otro reputado investigador, Jacobson, se unió a Rational y se incluyeron ideas suyas.

Estas tres personas son conocidas como los "tres amigos". Además, este lenguaje se abrió a la colaboración de otras empresas para que aportaran sus ideas. Todas estas colaboraciones condujeron a la definición de la primera versión de UML.

Desde hace unos años, las tecnologías de la información y comunicación ya han producido una enorme variedad de métodos y notaciones para llevar a cabo el modelado. Existen métodos y anotaciones para el diseño, la estructura, el procesamiento y el almacenamiento de información. De la misma manera también podemos encontrar métodos para la planificación, modelado, implementación, ensamblaje, prueba, documentación, ajuste, etc. de los sistemas. Entre los conceptos que se utilizan existen algunos relativamente fundamentales y, debido a eso, se expanden más allá del ámbito en el que fueron creados en un principio.

Desde la concepción de la tecnología de la información hasta finales de 1970, los desarrolladores de software se tomaron el desarrollo del software como un arte. Pero estos sistemas fueron poco a poco haciendose más complejos y por esta razón el mantenimiento y el desarrollo exigía otro tipo de visión, más allá del previamente descrito. Este hecho dio lugar a la ya famosa crisis del software, que lleva al enfoque de ingeniería (ingeniería de software) y la programación estructurada. Se desarrollaron métodos para la estructuración de sistemas y para los procesos de diseño, desarrollo y mantenimiento. Los enfoques orientados a procesos, por ejemplo, el método de salida de procesamiento de entrada de jerarquía, enfatizaron la funcionalidad de los sistemas. Con este método, el sistema total se divide en componentes más pequeños a través de la descomposición funcional.

En todos estos métodos y notaciones, dividimos el sistema en dos partes: una sección de datos y una sección de procedimientos. Esto es claramente reconocible en lenguajes de programación más antiguos, como COBOL. Los diagramas de flujo de datos, los diagramas de estructura, los diagramas HIPO y los diagramas de Jackson se utilizan para ilustrar el rango de funciones.

En la década de 1980, el análisis estructural clásico se desarrolló aún más. Los desarrolladores generaron diagramas de relaciones de entidades para el modelado de datos y redes de Petri para el modelado de procesos.

A medida que los sistemas se volvieron más complejos, ya no se podría diseñar cada sistema "desde cero". Las propiedades, como la mantenibilidad y la reutilización, se hicieron cada vez más importantes. Se desarrollaron lenguajes de programación orientados a objetos, y con ellos, los primeros lenguajes de modelado orientados a objetos surgieron en los años 70 y 80. En la década de 1990, las primeras publicaciones sobre análisis orientado a objetos y diseño orientado a objetos se pusieron a disposición del público. A mediados de la década de 1990, ya existían más de 50 métodos orientados a objetos, así como muchos formatos de diseño. Un lenguaje de modelado unificado parecía indispensable.

A principios de la década de 1990, los métodos orientados a objetos de Grady Booch y James Rumbaugh se utilizaron ampliamente. En octubre de 1994, Rational Software Corporation (parte de IBM desde febrero de 2003) comenzó la creación de un lenguaje de modelado unificado. Primero, acordaron una estandarización de la notación (lenguaje), ya que esto parecía menos elaborado que la estandarización de los métodos. Al hacerlo, integraron el Método Booch de Grady Booch, la Técnica de modelado de objetos (OMT) de James Rumbaugh y la Ingeniería de software orientada a objetos (OOSE), de Ivar Jacobsen, con elementos de otros métodos y publicaron esta nueva notación bajo el nombre UML, versión 0.9 .

El objetivo no era formular una notación completamente nueva, sino adaptar, expandir y simplificar los tipos de diagramas existentes y aceptados de varios métodos orientados a objetos, como los diagramas de clase, los diagramas de casos de uso de Jacobson o los diagramas de gráficos de estado de Harel. Los medios de representación que se utilizaron en los métodos

estructurados se aplicaron a UML. Por lo tanto, los diagramas de actividad de UML están, por ejemplo, influenciados por la composición de los diagramas de flujo de datos y las redes de Petri.

> *"Lo que es destacable en UML no es su contenido, sino su estandarización a un solo lenguaje unificado con un significado definido formalmente."*

Compañías conocidas, como IBM, Oracle, Microsoft, Digital, Hewlett-Packard y Unisys se incluyeron en el desarrollo posterior de UML. En 1997, la versión 1.1 de UML fue enviada y aprobada por la OMG.

¿Por qué modelamos?

Los proyectos de software que fracasan lo hacen por circunstancias propias, pero todos los proyectos con éxito se parecen en muchos aspectos. Entre todos los aspectos que hacen que un proyecto tenga éxito uno en común es el uso del modelado.

El modelado es una técnica de ingeniería probada y bien aceptada. Se construyen modelos arquitectónicos de casas y rascacielos para ayudar a sus usuarios a visualizar el producto final. Incluso podemos construir modelos matemáticos para analizar los efectos de vientos o terremotos sobre nuestros edificios.

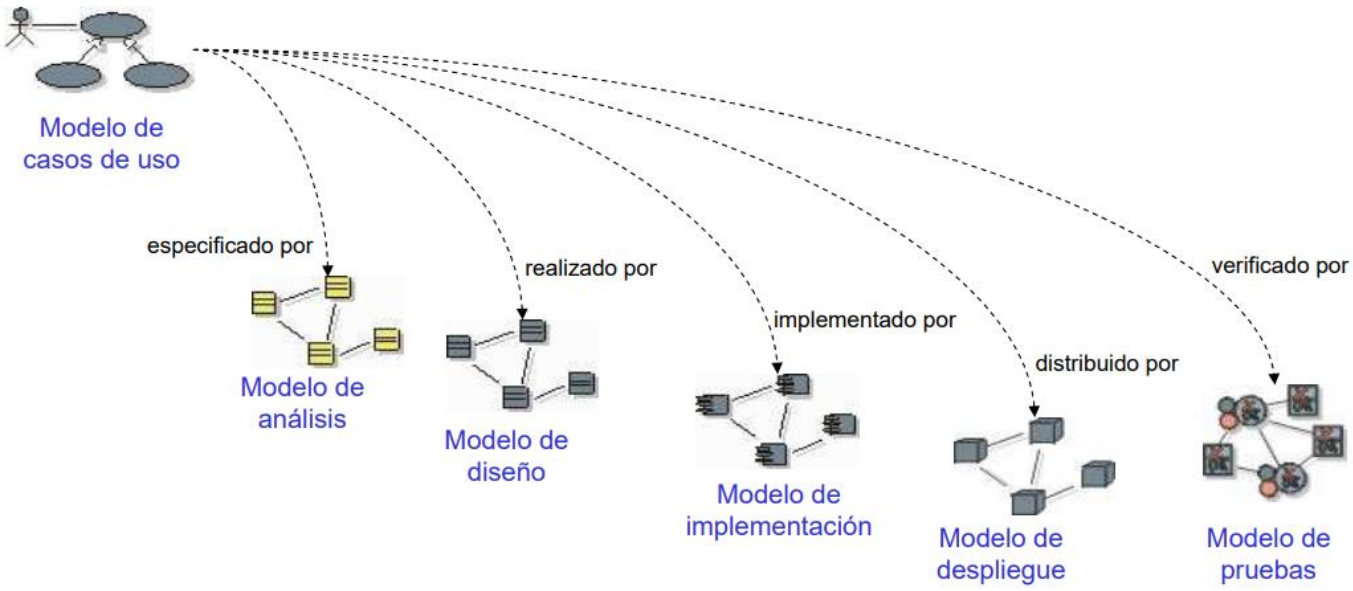
Un ***modelo*** es una simplificación de la realidad. Un modelo proporciona los ***planos*** de un sistema. Pueden ser planos detallados, así como planos más generales que ofrecen una visión global del sistema en consideración. Un buen modelo incluye aquellos elementos que tienen una gran influencia y omite aquellos menores que no son relevantes para el nivel de abstracción dado. Todo sistema puede ser descripto desde diferentes perspectivas utilizando diferentes modelos. Un modelo puede ser estructural resaltando la organización del sistema o puede ser de comportamiento, destacando su dinámica.

- Un modelo captura las propiedades estructurales (estática) y de comportamiento (dinámicas) de un sistema.
- Es una abstracción de dicho sistema, considerando un cierto propósito.
- El modelo describe completamente aquellos aspectos del sistema que son relevantes al propósito del modelo, y a un apropiado nivel de detalle.
- El código fuente del sistema es el modelo más detallado del sistema (y, además, es ejecutable). Sin embargo, se requieren otros modelos.
- Cada modelo es completo desde un punto de vista del sistema. Sin embargo, existen relaciones de trazabilidad entre los diferentes modelos.

Se construyen modelos para comprender mejor el sistema que estamos desarrollando. A través del modelado se persiguen los siguientes objetivos:

- Los modelos ayudan a visualizar cómo es lo que queremos que sea un sistema.
- Los modelos permiten especificar la estructura o el comportamiento de un sistema.
- Los modelos proporcionan plantillas que sirven de guía en la construcción de un programa.
- Los modelos documentan las decisiones que se han adoptado.
- Los modelos a utilizar los define la metodología que se aplique en el proceso de desarrollo.

Por ejemplo, con **PUD**:



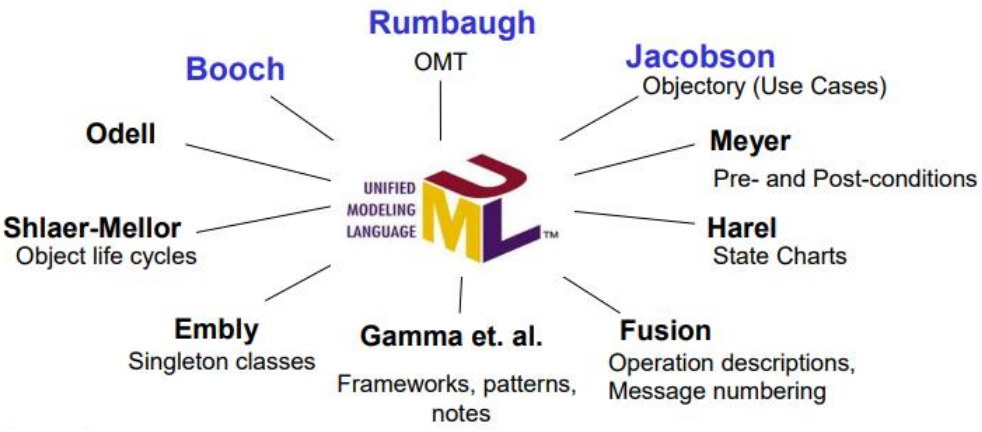
CLASE 1 - ¿Qué es UML?

El Lenguaje Unificado de Modelado (Unified Modeling Languaje, UML) en un lenguaje estándar para escribir "planos" de software. UML puede utilizarse para visualizar, especificar, construir y documentar los artefactos de un sistema que involucra una gran cantidad de software.

UML, como lo establecen sus autores, es apropiado para modelar desde sistemas de información en empresas hasta aplicaciones distribuidas basadas en la Web.

UML, es sólo un lenguaje (no es una *metodología*) y por lo tanto es sólo una parte de un método de desarrollo de software. UML es independiente del proceso, aunque para utilizarlo óptimamente, sus autores aconsejan utilizar un proceso que fuese dirigido por los casos de uso, centrado en la arquitectura, iterativo e incremental.

Agrupa notaciones y conceptos provenientes de distintos tipos de métodos O.O



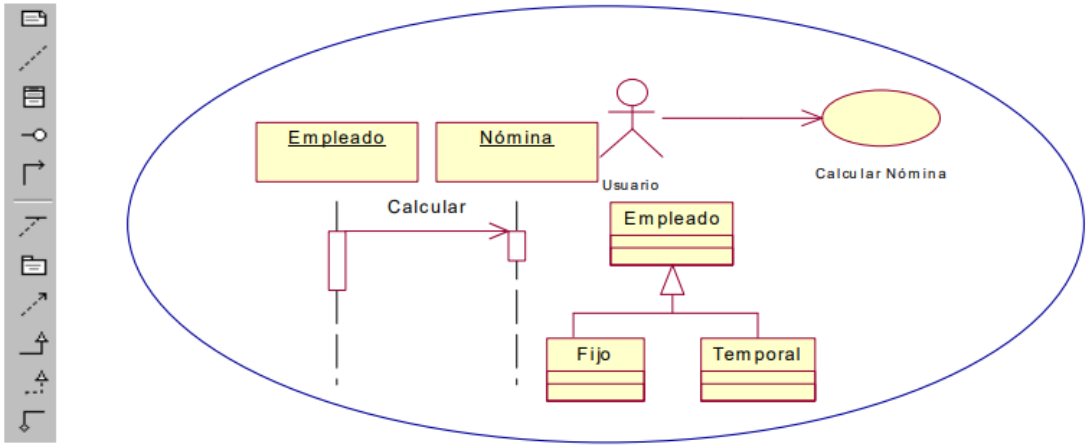
Propósito de UML

Ya mencionamos que UML en un ***lenguaje***. Un lenguaje proporciona un vocabulario y las reglas para combinar palabras de ese vocabulario con el objetivo de posibilitar la comunicación. Un lenguaje de modelado es un lenguaje cuyo vocabulario y reglas se centran en la representación conceptual y física de un sistema.

Propósito de UML

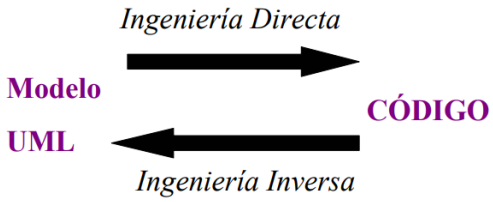
El modelado proporciona una comprensión de un sistema. Nunca es suficiente con un único modelo, a menudo se necesitan múltiples modelos conectados entre sí. El vocabulario y las reglas de UML indican cómo crear y leer modelos bien formados, pero no dicen qué modelos se deben crear ni cuándo se deberían crear. Así, UML es un lenguaje para:

- **Visualizar:** Un modelo explícito facilita la comunicación. Algunas cosas se modelan mejor textualmente, otras se modelan mejor de forma gráfica. En todos los sistemas interesantes hay estructuras que trascienden lo que puede ser representado en un lenguaje de programación. UML es uno de estos lenguajes gráficos, es algo más que un conjunto de símbolos gráficos. Detrás de cada símbolo en la notación UML hay una semántica bien definida. Así, un desarrollador puede construir un modelo en UML y otro desarrollador o incluso otra herramienta, puede interpretar ese modelo sin ambigüedad.



- **Especificar:** En este contexto especificar es construir modelos precisos, no ambiguos y completos. UML cubre la especificación de todas las decisiones de análisis, diseño e implementación que deben realizarse al desarrollar y desplegar un sistema con gran cantidad de software.

- **Construir:** UML no es un lenguaje de programación visual, pero sus modelos pueden conectarse de forma directa a una gran variedad de lenguajes de programación. Esto significa que es posible hacer correspondencias desde un modelo UML a un lenguaje de programación como Java, C++ o Visual Basic, incluso a tablas en una base de datos relacional o al almacenamiento persistente de una base de datos orientada a objetos. Esta correspondencia permite ingeniería directa: la generación de código a partir de un modelo UML en un lenguaje de programación.



- **Documentar:** Una organización de software que trabaje bien produce toda clase de artefactos además de código ejecutable. Estos artefactos incluyen, entre otros: requisitos, arquitectura, diseño, código fuente, planificación de proyectos, pruebas, prototipos, versiones.

Bloques de Construcción

El vocabulario de UML incluye tres clases de bloques de construcción: ***elementos, diagramas y relaciones***.

Elementos en UML: Hay 4 tipos de elementos en UML, elementos estructurales, elementos de comportamiento, elementos de agrupación y elementos de anotación:

Elementos estructurales:

Por ejemplo: ***clases, interfaces, colaboraciones, casos de uso, clases activas, componentes y nodos***. Son los elementos estructurales básicos que se pueden incluir en el modelo UML. También existen variaciones de estos siete elementos, tales como actores, procesos, hilos y aplicaciones, documentos, archivos, bibliotecas, páginas y tablas.

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 en el espacio. Se clasifican en: interacción y máquina de estados. Estos dos elementos (interacciones y máquinas de estado) son los elementos de comportamiento que se pueden incluir de un modelo UML, estos elementos están conectados normalmente a diversos elementos estructurales, principalmente clases, colaboraciones y objetos.

- ***Interacción***: Es un comportamiento que comprende un conjunto de mensajes intercambiados entre un conjunto de objetos, dentro de un contexto particular para alcanzar un propósito específico.
- ***Máquina de Estados***: Es un comportamiento que especifica las secuencias de estados por las que pasa un objeto o una interacción durante su vida en respuesta a eventos.

Los elementos de agrupación:

Son las partes organizativas de los modelos UML. Estos son las cajas en las que pueden descomponerse un modelo. Pueden ser paquetes.

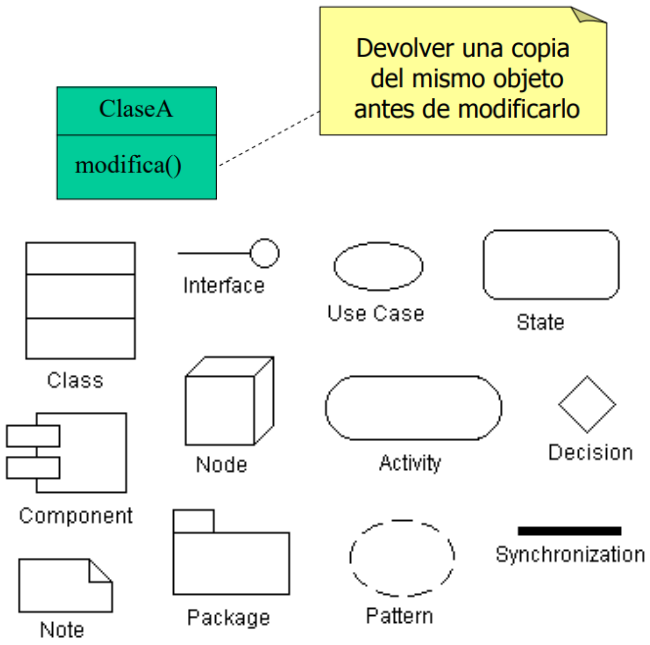
- ***Paquete***: es un mecanismo de propósito general para organizar elementos en grupos. En los paquetes se pueden agrupar los elementos estructurales, de comportamiento e incluso otros elementos de agrupación. Los paquetes son los elementos de agrupación básicos con los cuales se puede organizar un modelo UML. También hay variaciones, tales como los framework, los modelos y los subsistemas.

Los elementos de anotación:

Son las partes explicativas de los modelos UML.

- Son comentarios que se añaden para describir, clarificar y hacer observaciones.
- ***Hay un tipo principal***: Nota: Símbolo para mostrar restricciones y comentarios asociados a un elemento o colección de elementos. Se usan para aquello que se muestra mejor en forma textual (comentario) o formal (restricción).
-

Ejemplo:



Estereotipos:

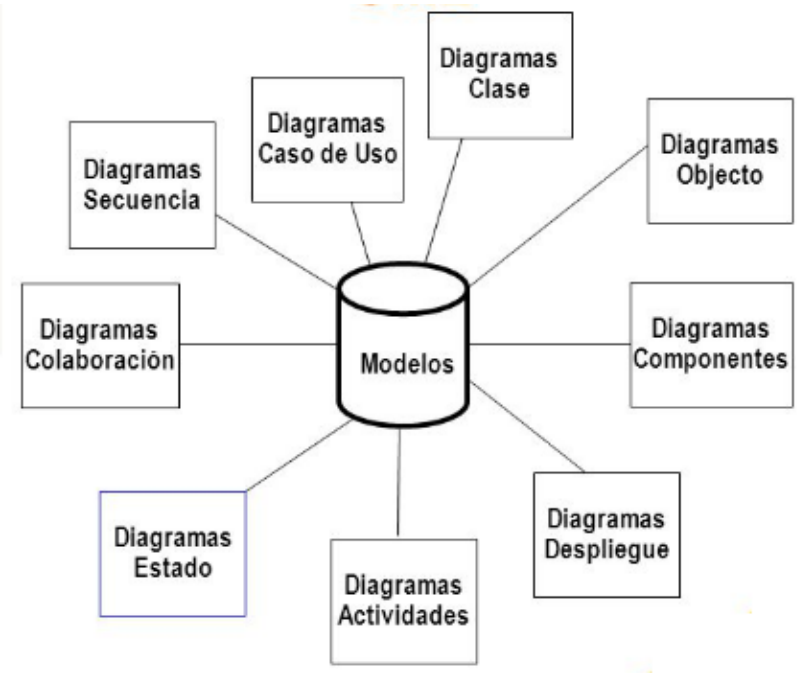
Algunos sistemas requieren de elementos hechos a medida que no se encuentran en el UML. Para ello, los estereotipos o clisés le permiten tomar elementos propios del UML y convertirlos en otros que se ajusten a las necesidades. Se representan como un nombre entre dos pares de paréntesis angulares. <<nombre>>

Diagramas

El UML está compuesto por diversos elementos gráficos que se combinan para conformar diagramas. Debido a que el UML es un lenguaje, cuenta con reglas para combinar tales elementos y la finalidad de los diagramas es presentar diversas perspectivas de un sistema, a las cuales se les conoce como modelo. (Recordemos que un modelo es una representación simplificada de la realidad; el modelo UML describe lo que supuestamente hará un sistema, pero no dice cómo implementar dicho sistema.)

Los diagramas de UML sirven para visualizar un sistema desde diferentes perspectivas.

- Un diagrama es una proyección gráfica de un sistema con una vista resumida de los elementos que constituyen el sistema.

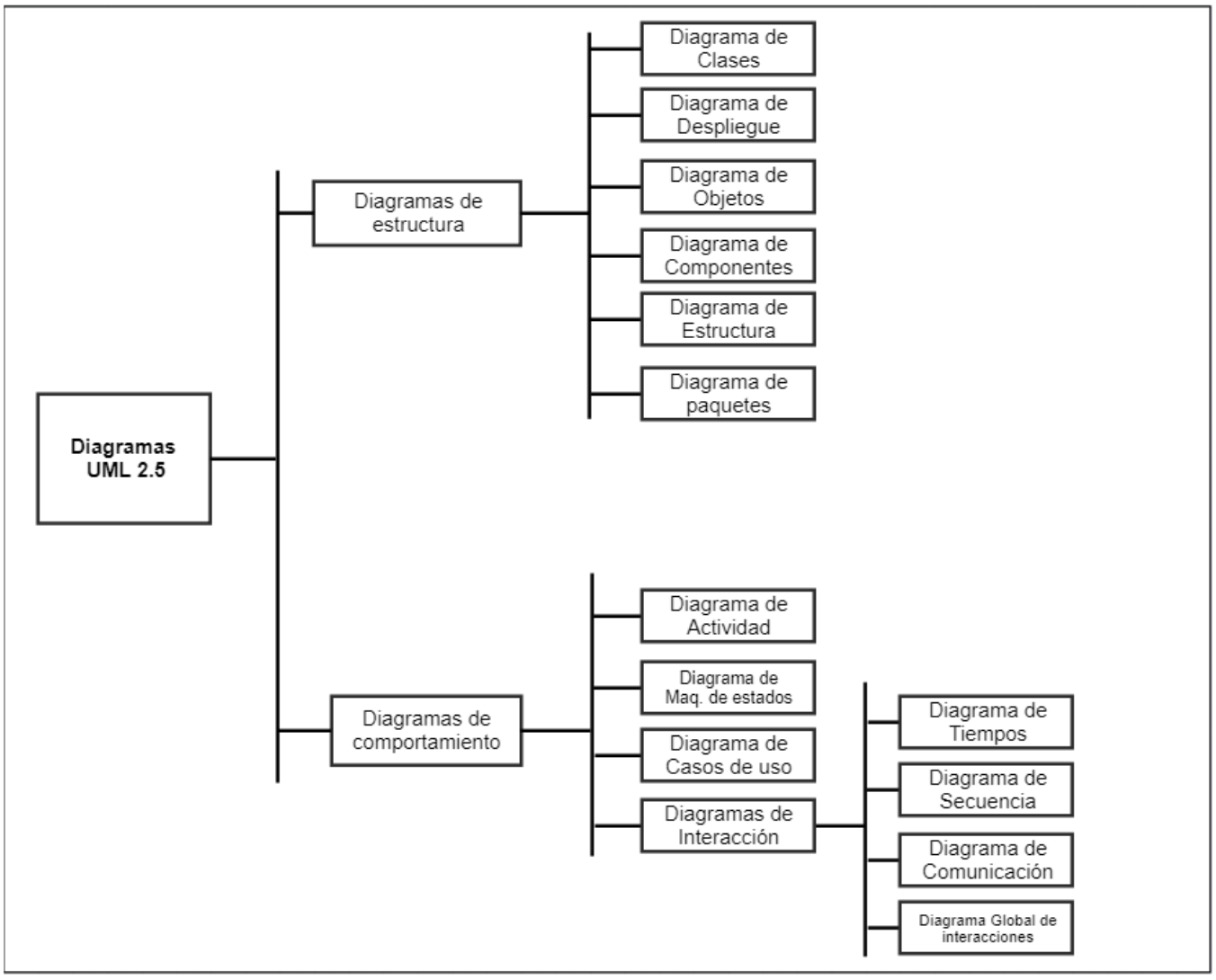


UML 2 incluye 13 tipos de diagramas.

Estructurales (Estática	De Comportamiento (Dinámica)

• Clases
• Objetos
• Componentes
• Despliegue
• Paquetes
• Estructura Compuesta
	• Caso de uso
	• Estados
	• Actividades
	• Iteración
	  ○ Secuencia
	  ○ Comunicación
	  ○ Tiempos
	  ○ Revisión de Iteraciones

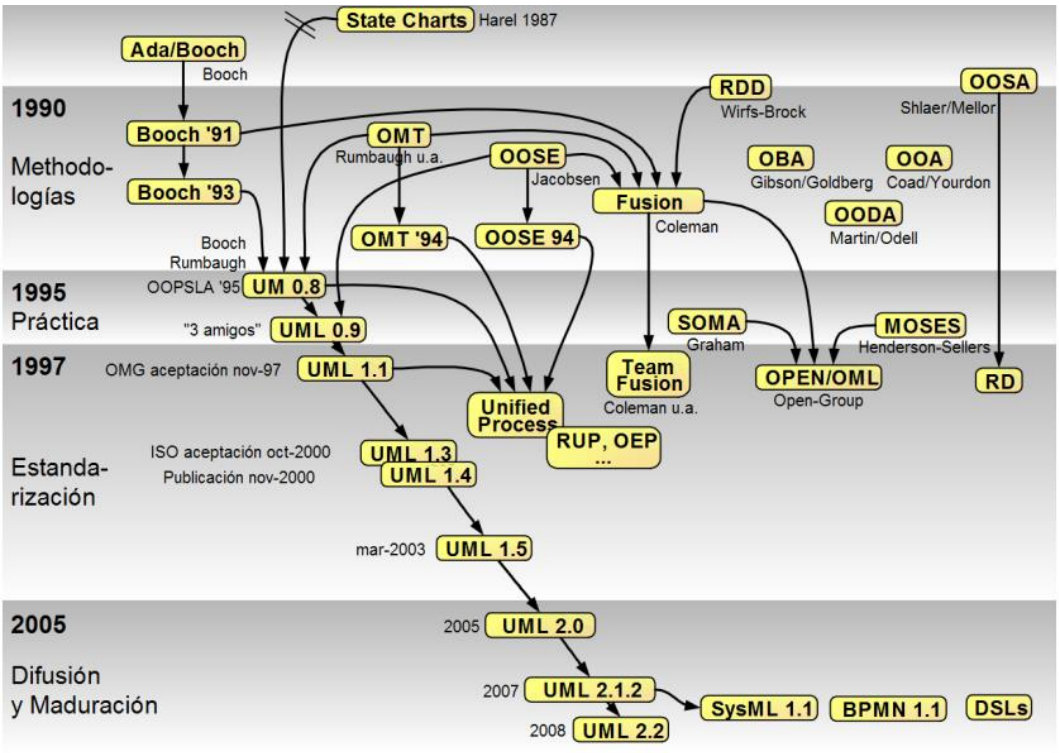
Actualmente, en la versión 2.5.1 de UML, existen dos clasificaciones de diagramas: Los **diagramas estructurales** y los **diagramas de comportamiento**. Todos los diagramas UML están contenidos en esta clasificación.



¿Qué versiones existen de UML?

La versión actual de UML es la 2.5.1 y fue publicada en junio de 2015. UML es gestionada y actualizada por la OMG (Objeto Manabement Group). Esta es la lista de versiones que han sido publicadas:

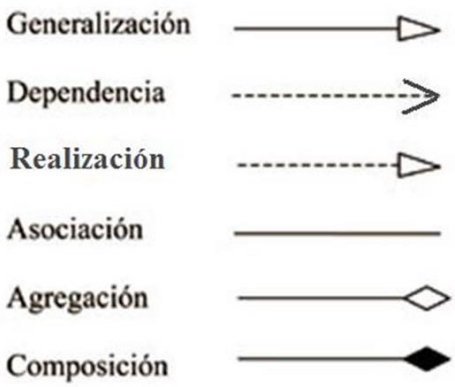
1.1 – Noviembre de 1997
1.3 – Marzo de 2000
1.4 – Septiembre de 2001
1.5 – Marzo de 2003
1.4.2 – Enero de 2005
2.0 – Octubre de 2005
2.1 – Abril de 2006
2.1.1 – Febrero de 2007
2.1.2 – Noviembre de 2007
2.2 – Febrero de 2009
2.3 – Mayo de 2010
2.4.1 – Agosto de 2011
2.5 – Junio de 2015
2.5.1 – Diciembre de 2017 (Última versión)



Relaciones

Cuando diferentes objetos trabajan en función a una meta o propósito y se logra por medio de la colaboración entre las partes a través de las relaciones se dice que existe una relación. En pocas palabras se define como la conexión entre elementos.

Para diferenciar las distintas relaciones se utilizan una Simbología con base en varios tipos de líneas.



Vamos a mencionar algunas definiciones de los tipos de relaciones que se pueden representar con UML

Asociación

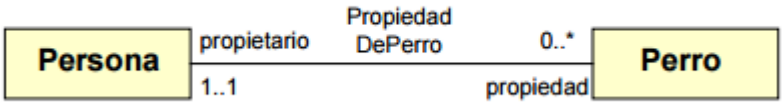
Las asociaciones representan las relaciones más generales entre clases, es decir, las relaciones con menor contenido semántico. Para UML una asociación va a describir un conjunto de vínculos entre las instancias de las clases.

Las asociaciones pueden ser binarias (conectan dos clases) o n-arias (conectan n clases), aunque lo más normal en un modelo es utilizar sólo relaciones binarias (en general, y sin entrar en detalles, se puede afirmar que una relación n-aria puede modelarse mediante un conjunto finito de relaciones binarias).

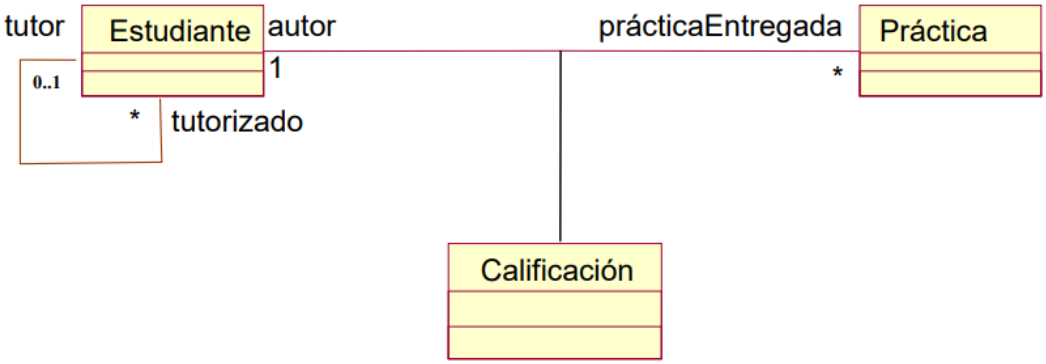
La forma de representar las asociaciones binarias en UML es mediante una línea que conecta las dos clases. En general, las asociaciones son bidireccionales, esto es, no tienen un sentido asociado. Más adelante, podremos ver que se puede "restringir" una relación de asociación indicando el sentido y dirección de la relación.

Ejemplo: Si tenemos la clase perro y persona las siguientes relaciones podrían darse:

La cual muestra que una persona es propietaria de uno o varios perros, pero estos son solo de esta persona.



Ejemplo de clase de asociación y asociación reflexiva.



Agregación y Composición: sirve para modelar objetos complejos en base a relaciones ***todo – parte***.

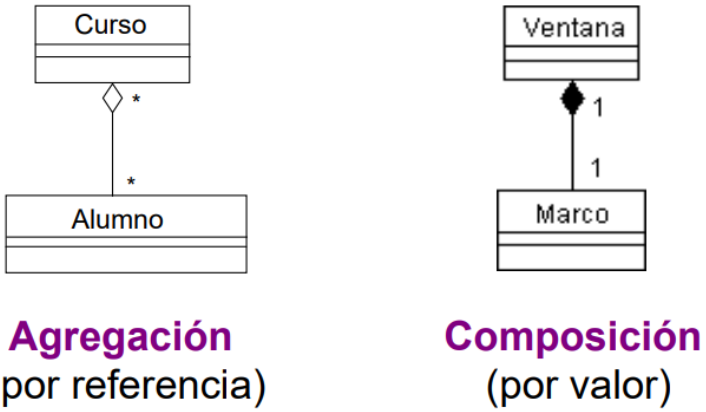
Agregación

- Es una Relación dinámica: el tiempo de vida del objeto que se agrega es independiente del objeto agregador.
- El objeto agregado utiliza al agregado para su funcionamiento.
- SIMILAR parámetro pasado "por referencia".

Composición

- Es una relación estática: el tiempo de vida del objeto incluido está condicionado por el tiempo de vida del objeto compuesto.
- El objeto compuesto se construye a partir del objeto incluido.
- SIMILAR parámetro pasado "por valor".

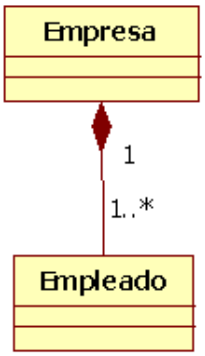
Ejemplo:



Composición

La composición implica que los componentes de un objeto sólo pueden pertenecer a un solo objeto agregado, de forma que cuando el objeto agregado es destruido todas sus partes son destruidas también.

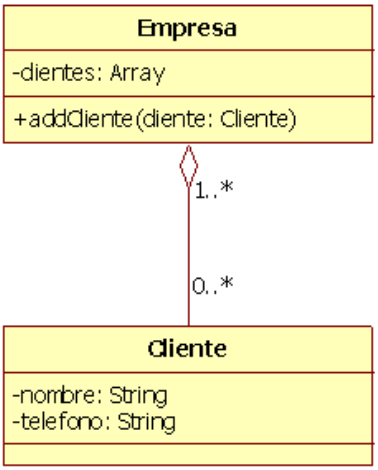
Ejemplo: A una empresa la componen empleados.



Agregación

La agregación es una asociación con unas connotaciones semánticas más definidas: la agregación es la relación parte-de, que presenta a una entidad como un agregado de partes (en orientación a objeto, un objeto como agregado de otros objetos).

Ejemplo: Una empresa tiene clientes

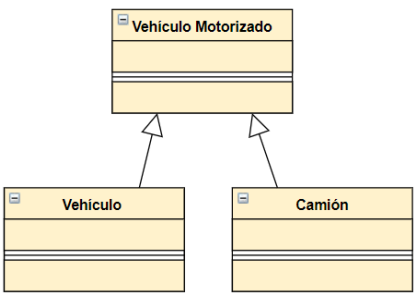


La agregación es un tipo especial de asociación, que representa una relación estructural entre todo y sus partes. Donde representa Una relación del tipo “Tienen-un” ejemplo: La empresa tiene un Cliente.

Generalización

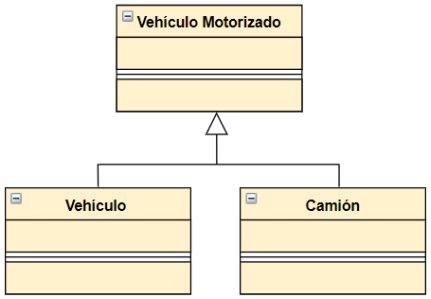
La generalización es la típica relación de herencia/especialización entre clases. En UML la herencia se representa mediante una flecha, cuya punta es un triángulo vacío. La flecha que representa a la generalización va orientada desde la subclase a la superclase.

Cuando de una superclase se derivan varias subclases existen dos notaciones diferentes, aunque totalmente equivalentes, para su representación.

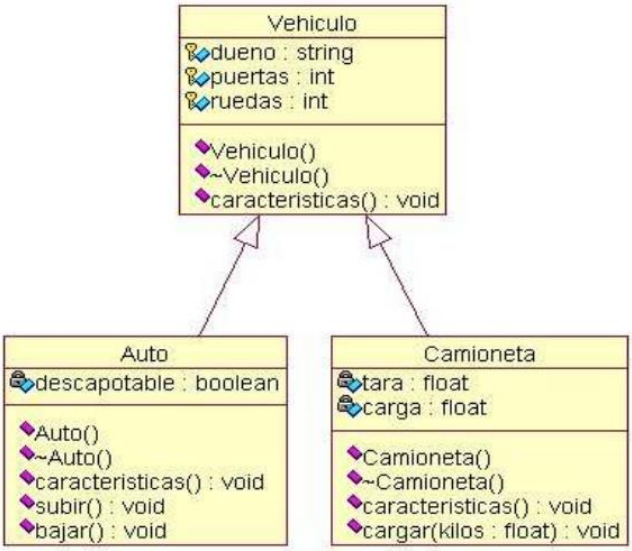


En la primera forma de representar esta situación se muestra una superclase a la que llegan tantas flechas como clases derivadas tiene.

En la segunda representación se tiene una única punta de flecha que llega a la superclase, pero a la base del triángulo que hace de punta de flecha lleguen tanto cominos como subclases haya.



Una generalización da lugar al polimorfismo entre clases de una jerarquía de generalizaciones: Un objeto de una subclase puede sustituir a un objeto de la superclase en cualquier contexto. Lo inverso no es cierto.



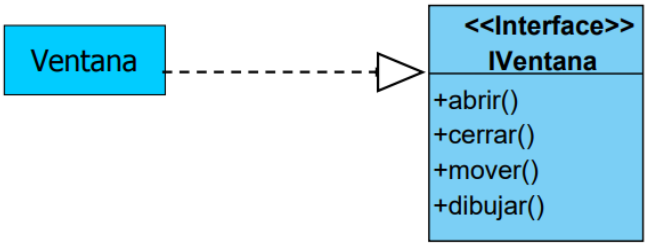
Realización

Es una relación semántica entre clasificadores donde este especifica unas normas o un reglamento con otro clasificador que garantiza que se cumplirá. Se encuentran relaciones de realización: entre interfaces, clases y componentes que las realizan y entre los casos de uso y las colaboraciones que los realizan.

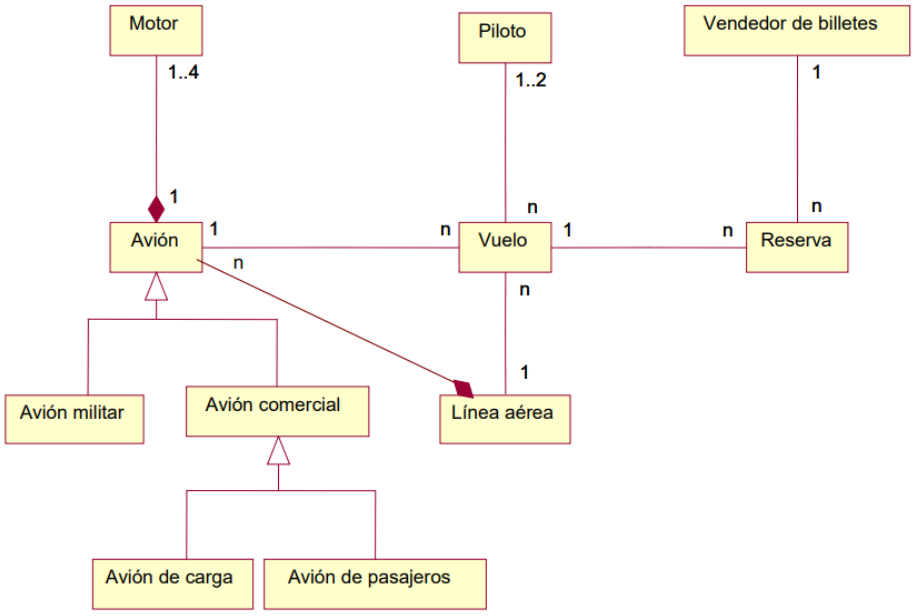
Relación semántica entre clasificadores, donde un clasificador especifica un contrato que otro clasificador garantiza que cumplirá.

Se pueden encontrar en dos casos:

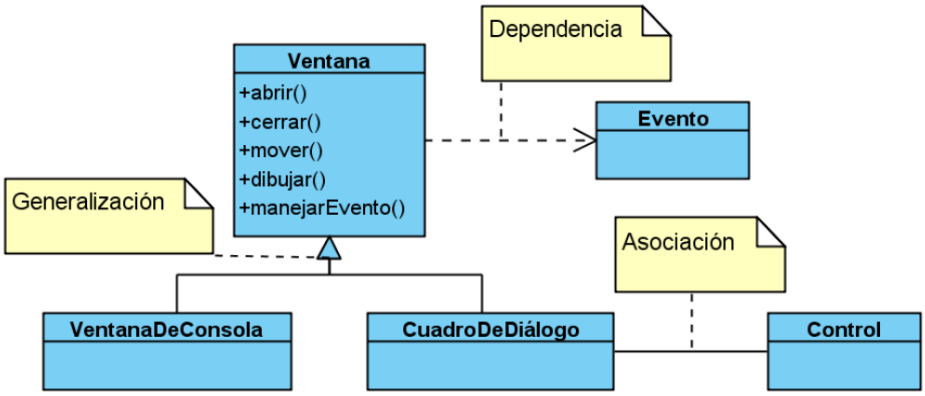
1 Clases o componentes que realizan interfaces.
2 Colaboraciones que realizan casos de uso.



Jerarquías de Relaciones



Ejemplo con generalizaciones, asociación y dependencia.



Vistas de un sistema con UML

Todos hemos visto muchos libros y artículos donde se intenta capturar todos los detalles de la arquitectura de un sistema usando un único diagrama. Pero si miramos cuidadosamente el conjunto de cajas y flechas que muestran estos diagramas, resulta evidente que sus autores han trabajado duramente para intentar representar más de un plano que lo que realmente podría expresar la notación. ¿Es acaso que las cajas representan programas en ejecución? ¿O representan partes del código fuente? ¿O computadores físicos? ¿O acaso meras agrupaciones de funcionalidad? ¿Las flechas representan dependencias de compilación? ¿O flujo de control? Generalmente es un poco de todo.

¿Será que una arquitectura requiere un estilo único de arquitectura? A veces la arquitectura del software tiene secuelas de un diseño del sistema que fue muy lejos en particionar prematuramente el software, o de un énfasis excesivo de algunos de los aspectos del desarrollo del software: ingeniería de los datos, o eficiencia en tiempo de ejecución, o estrategias de desarrollo y organización de equipos. A menudo la arquitectura tampoco aborda los intereses de todos sus "clientes". El modelo de 4+1 vistas fue desarrollado para remediar este problema.

El modelo de vistas 4+ 1

El modelo "4+1" de Kruchten, es un modelo de vistas diseñado por el profesor Philippe Kruchten y que encaja con el estándar "IEEE 1471-2000" (Recommended Practice for Architecture Description of Software-Intensive Systems) que se utiliza para describir la arquitectura de un sistema software intensivo basado en el uso de múltiples puntos de vista.

Este modelo lo que propone Kruchten es que un sistema software se ha de documentar y mostrar (tal y como se propone en el estándar IEEE 1471-2000) con 4 vistas bien diferenciadas y estas 4 vistas se han de relacionar entre sí con una vista más, que es la denominada vista "+1". Estas 4 vista las denominó Kruchten como: *vista lógica, vista de procesos, vista de despliegue y vista física y la vista "+1"* que tiene la función de relacionar las 4 vistas citadas, la denominó vista de escenario.

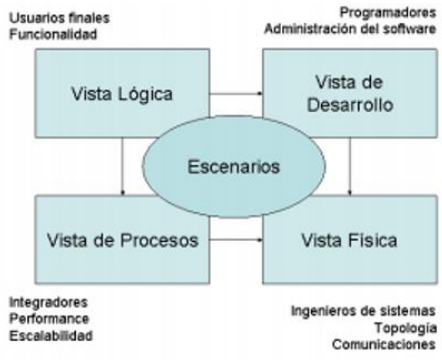
La arquitectura del software se trata de abstracciones, de descomposición y composición, de estilos y estética. También tiene relación con el diseño y la implementación de la estructura de alto nivel del software. Los diseñadores construyen la arquitectura usando varios elementos arquitectónicos elegidos apropiadamente. Estos elementos satisfacen la mayor parte de los requisitos de funcionalidad y performance del sistema, así como también otros requisitos no funcionales tales como confiabilidad, escalabilidad, portabilidad y disponibilidad del sistema.

Durante el desarrollo de un sistema software se requiere que éste sea visto desde varias perspectivas.

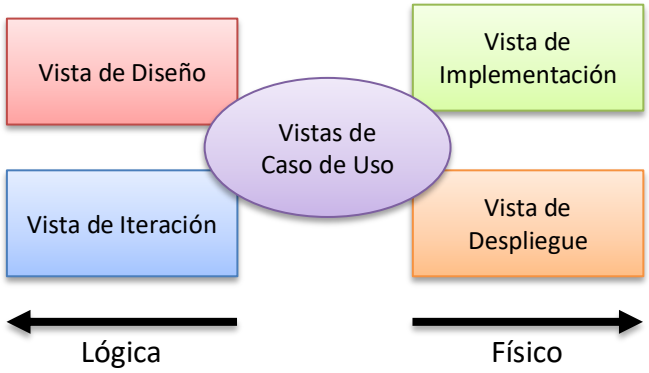
- Diferentes usuarios miran el sistema de formas diferentes en momentos diferentes.
- La arquitectura del sistema es clave para poder manejar estos puntos de vista diferentes y se organiza mejor a través de vistas arquitecturales interrelacionadas.

Entendemos por *arquitectura* el conjunto de decisiones significativas sobre:

- La organización de un sistema de software.
- La selección de elementos estructurales y sus interfaces.
- Su comportamiento (colaboraciones entre esos elementos).
- La composición de los elementos estructurales y de comportamiento en subsistemas cada vez más grandes.
- El estilo arquitectónico que guía esta organización (los elementos estáticos y dinámicos y sus interfaces, colaboraciones y su composición).



La arquitectura de un sistema se puede describir a través de cinco vistas relacionadas entre sí, como se muestra en el siguiente gráfico:

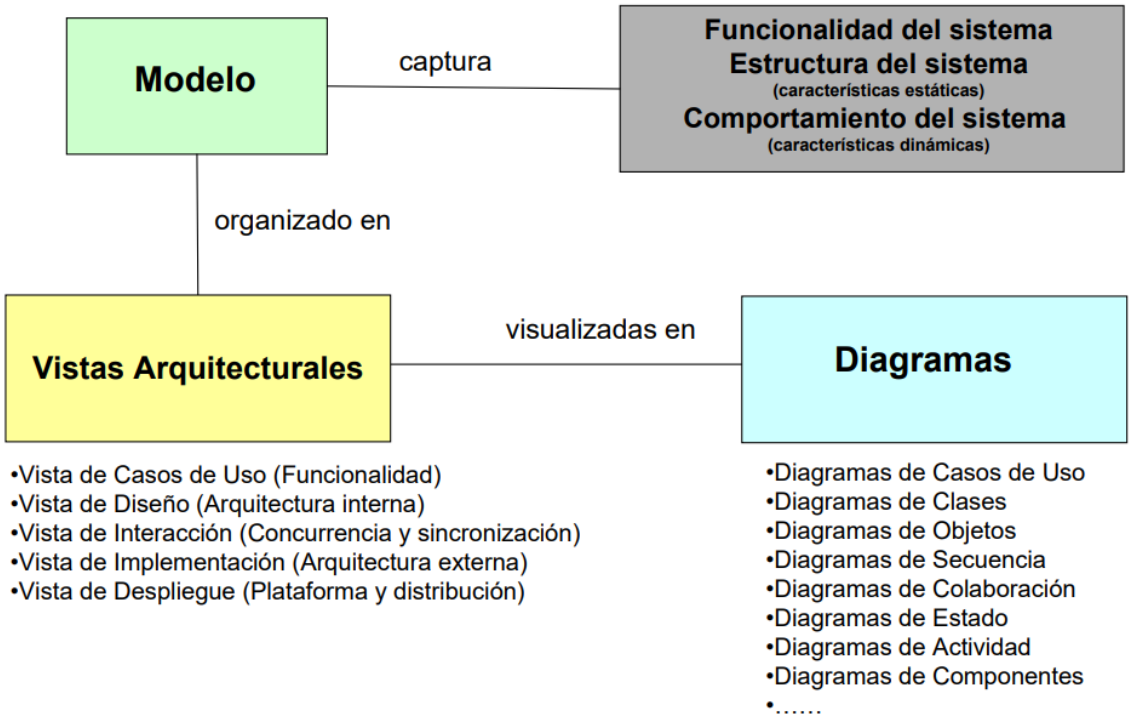


Vista de casos de uso	Describe el comportamiento del sistema a través de los casos de uso.	• ***Aspecto estático***: Diagrama de casos de uso. • ***Aspecto dinámico***: Diagramas de interacción, de estados y de actividades.
Vista de diseño	Comprende las clases, interfaces y colaboraciones que conforman el vocabulario del problema y la solución. Soporta principalmente los requisitos funcionales del sistema.	• Aspecto estático: Diagramas de clases y objetos. • Aspecto dinámico: Diagramas de interacción, de estados y de actividades.
Vista de procesos	Comprende los hilos y procesos que forman los mecanismos de sincronización y concurrencia del sistema. En esta vista se muestran los procesos que hay en el sistema y la forma en la que se comunican estos procesos; es decir, se representa desde la perspectiva de un integrador de sistemas, el flujo de trabajo paso a paso de negocio y operacionales de los componentes que conforman el sistema. Para completar la documentación de esta vista se puede incluir el diagrama de actividad de UML.	Los aspectos estáticos y dinámicos de esta vista se capturan con los mismos diagramas que la vista de diseño, pero poniendo énfasis en las clases activas. • ***Aspecto estático***: Diagramas de clases y objetos. • ***Aspecto dinámico***: Diagramas de interacción, de estados y de actividades.
Vista de implementación	Comprende los componentes y archivos que se utilizan para ensamblar y hacer disponible el sistema físico. En esta vista se muestra desde la perspectiva de un ingeniero de sistemas todos los componentes físicos del sistema, así como las conexiones físicas entre esos componentes que conforman la solución (incluyendo los servicios). Para completar la documentación de esta vista se puede incluir el diagrama de despliegue de UML.	• Aspecto estático: Diagramas de componentes. • ***Aspecto dinámico***: Diagramas de interacción, de estados y de actividades.
Vista de despliegue	En esta vista se muestra el sistema desde la perspectiva de ***un programador*** y se ocupa de la gestión del software; o, en otras palabras, se va a mostrar cómo está dividido el sistema software en componentes y las dependencias que hay entre esos componentes. Para completar la documentación de esta vista se pueden incluir los diagramas de componentes y de paquetes de UML.	• ***Aspecto estático***: Diagramas de despliegue • ***Aspecto dinámico***: Diagramas de interacción, de estados y de actividades.

En resumen

Cada vista puede existir de forma independiente, pero interactúan entre sí:

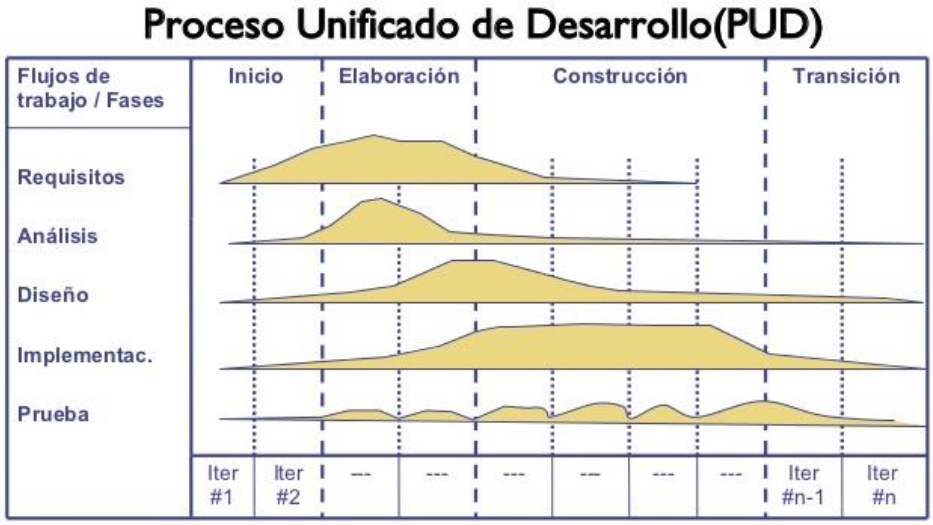
Los nodos (vista de despliegue) contienen componentes (vista de implementación). Dichos componentes representan la realización (software real) de las clases, interfaces, colaboraciones y clases activas (vistas de diseño y de interacción). Dichos elementos de las vistas de diseño e interacción representan el sistema solución a los casos de uso (vista de casos de uso) que expresan los requisitos.



CLASE 2 – Procesos Unificado de Desarrollo - P.U.D

El objetivo del Proceso Unificado de Desarrollo, es establecer un marco de trabajo para desarrollar software de calidad y con rigor. Este modelo de proceso se asienta en un conjunto subyacente de filosofías y principios para conseguir un desarrollo de software con una infraestructura de bloques de construcción del proceso y de contenidos reutilizables y presenta las siguientes características principales:

- Está dirigido por los llamados "casos de uso"
- Es centrado en la "arquitectura", como espina dorsal del software
- Es iterativo e incremental



El PUD tiene 4 fases:

- ***Fase de inicio***: se desarrolla una descripción del producto final a partir de una idea y se presenta el análisis de negocio para el producto.
- ***Fase de elaboración***: se especifican en detalle la mayoría de los CU y se diseña la arquitectura del sistema.
- ***Fase de construcción***: se crea al producto.
- ***Fase de transición***: se prueba el producto y se informan defectos e deficiencias.

Flujos de trabajo del PUD

Un FT es un conjunto de actividades, herramientas y estándares que se usan en un proceso. Cada FT está compuesto por:

- ***Actividades***: son las tareas que se llevan a cabo en el FT.
- ***Trabajadores***: son quiénes ejecutan las actividades.
- ***Artefactos***: es toda la información generada a partir del trabajo realizado en el FT.

En el PUD existen cinco FT:

1. FT de Requisitos.
2. FT de Análisis.
3. FT de Diseño
4. FT de Implementación.
5. FT de Prueba.

Flujo de Trabajo de Requisitos

El propósito fundamental del flujo de trabajo de requisitos es guiar el desarrollo hacia el sistema correcto. Esto se consigue mediante una descripción, suficientemente buena, de los requisitos del sistema como para que los desarrolladores lleguen a un acuerdo con el cliente sobre qué debe hacer y qué no debe hacer el sistema. Hay una serie de pasos que están siempre presentes:

- Enumerar los requisitos candidatos:
  - Encontrar requisitos funcionales
  - Encontrar requisitos no funcionales
- Comprender el contexto del sistema: Para capturar los requisitos correctos, que nos lleven a construir el sistema correcto, se requiere un firme conocimiento del contexto en el que se emplaza el sistema. Hay, por lo menos, dos formas de expresar el contexto de un sistema en una forma que sea de utilidad para los desarrolladores del sistema: el modelado del dominio y el modelado del negocio.

Modelado del negocio

El modelado del negocio se realiza para describir los procesos del negocio, con una visión orientada a objetos, con el objetivo de comprenderlos. El modelado de negocio es parte de la Ingeniería de Negocios, que es una técnica que tiene por objetivo mejorar los procesos de negocio de la organización y no forma parte del programa de esta asignatura.

Modelado del dominio

El modelo de dominio, también conocido como “Modelo de Objetos del Dominio del Problema”, describe los conceptos importantes del contexto como objetos del dominio y enlaza estos objetos entre sí.

Un modelo de dominio captura los tipos de objetos más importantes en el contexto del sistema. Los objetos del dominio representan las “cosas” que existen o los “eventos” que suceden en el entorno en el que se desenvuelve el sistema.

Muchos de las clases del dominio del problema, pueden deducirse de la especificación de requisitos o mediante la entrevista con los expertos del dominio. Las clases del dominio aparecen como:

- Entidades del negocio que representan cosas que se manipulan en el negocio como pedidos, cuentas, contratos.
- Entidades del mundo real y conceptos de los que el sistema debe hacer un seguimiento, como artículos, vehículos.
- Sucesos que ocurrirán o han ocurrido, como la partida y/o llegada de un avión, un siniestro en una compañía de seguros.

El modelado del dominio debe contribuir a la comprensión del problema que se supone que el sistema resuelve. Para desarrollar este modelo debe conformarse un equipo en el que estén presentes tanto los expertos del dominio (usuarios) como gente con experiencia en modelado, coordinado por los analistas del dominio.

El modelo de dominio ayuda a los usuarios, clientes, desarrolladores y otros participantes del proyecto a utilizar un vocabulario común. La terminología común es necesaria para compartir el conocimiento con los otros.

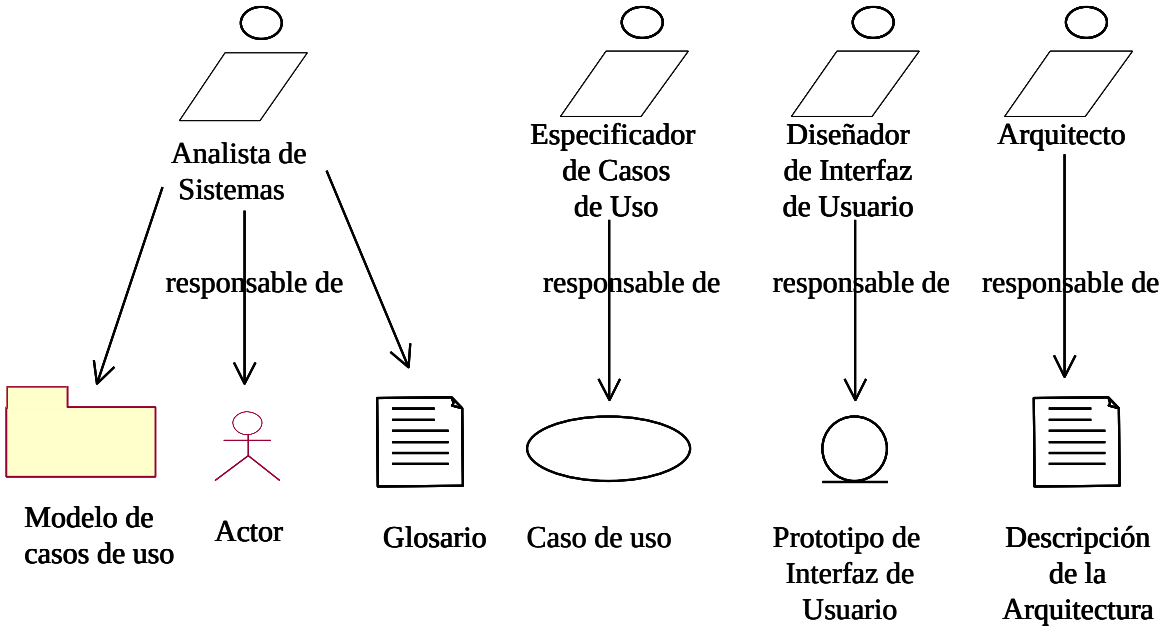
Las CLASES del dominio que aquí se encuentren se utilizarán al describir los casos de uso y diseñar el prototipo de interfaz de usuario y para sugerir clases internas al sistema en desarrollo durante el Análisis.

Para describir el dominio del problema se utilizan dos diagramas de UML, el **Diagrama de clases y el Diagrama de casos de uso.**

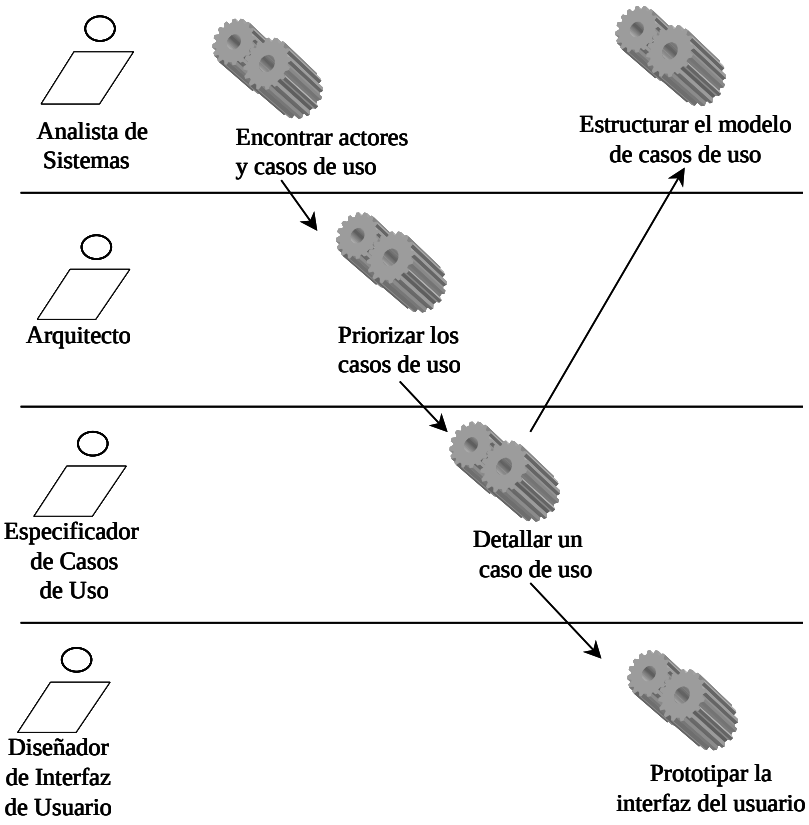
Para describir el flujo de trabajo de los requisitos, mencionaremos los artefactos creados en este flujo de trabajo, los trabajadores participantes y las actividades a realizar:

- **Artefactos**: Los artefactos fundamentales que se utilizan en la captura de requisitos son:
  - Modelo de Casos de Uso, que incluye los casos de uso y los actores.
  - Descripción de la arquitectura
  - Glosario
  - Prototipo de la interfaz del usuario
- **Trabajadores**: Los trabajadores responsables por los artefactos que se producen en el modelado de casos de uso son:
  - Analista de Sistemas.
  - Especificador de casos de uso.
  - Diseñador de interfaz de usuario.
  - Arquitecto.
- **Actividades:** Las actividades a realizar por los trabajadores para producir los distintos artefactos son:
  - Encontrar actores y casos de uso.
  - Priorizar casos de uso.
  - Detallar los casos de uso.
  - Prototipar la interfaz del usuario.
  - Estructurar el modelo de casos de uso.

El siguiente gráfico muestra la relación entre los artefactos del modelado de casos de uso y los trabajadores responsables de cada uno:



A continuación, se muestra un gráfico que indica el flujo de trabajo para la captura de requisitos en forma de casos de uso, incluyendo los trabajadores participantes y sus actividades:



Modelando clases del dominio del problema

Diagrama de Clases de objetos del dominio del problema (M.O.D.P)

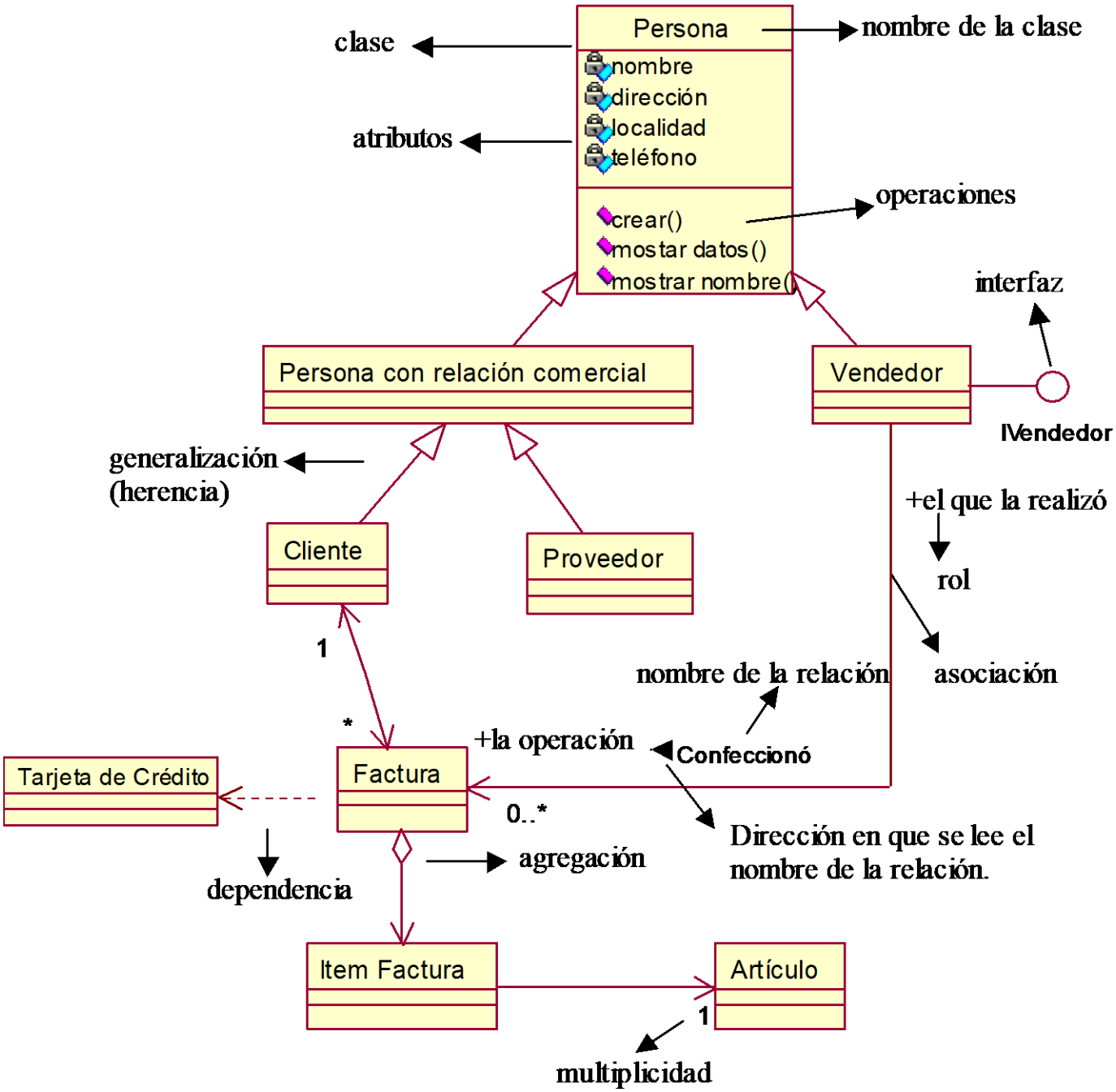
El diagrama de clases es un diagrama que muestra un conjunto de clases, interfaces, sus colaboraciones y relaciones. Se utilizan para modelar la vista de diseño estática de un sistema. Con UML los diagramas de clases se emplean para visualizar el aspecto estático de los bloques de construcción básicos del sistema y sus relaciones, y para especificar los detalles para construirlos.

En el diagrama de clases podemos encontrar:

- Clases
- Relaciones: herencia, asociación (su variante: agregación), dependencia.
- Indicadores de multiplicidad y navegabilidad
- Nombre de Rol

El diagrama de clases se puede representar gráficamente en forma abreviada, cuando no muestra atributos ni métodos o de forma completa en caso de mostrarlos.

Simbología del Diagrama de Clases



Clases

Es una plantilla para la creación de objetos de datos según un modelo predefinido. Las clases se utilizan para representar entidades o conceptos.

Relaciones

Un objeto por sí mismo no es demasiado interesante. Los objetos contribuyen al comportamiento de un sistema colaborando con otros. Hay dos tipos de relaciones que se pueden dar entre objetos: enlaces y agregación.

Enlaces

El término *enlace* es definido por Rumbaugh como "una conexión física o conceptual entre objetos". Un objeto colabora con otros a través de sus enlaces con éstos. Esto quiere decir que un enlace denota la asociación específica por la cual un objeto (el cliente) utiliza los servicios de otro objeto (el servidor) o a través de la cual un objeto puede comunicarse con otro.

Un concepto esencial en el paradigma orientado a objetos es el hecho de que los objetos "colaboran" entre sí para llevar a cabo un comportamiento superior.

Colaboración

Una *colaboración* representa la solicitud de un cliente a un servidor para cumplir una responsabilidad del cliente. Es la representación de un contrato entre un cliente y un servidor. Los objetos de una clase pueden cumplir una responsabilidad particular solos o pueden requerir la asistencia de otros objetos (de otras clases).

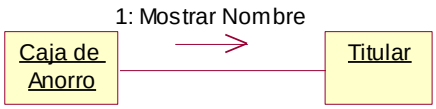
Se dice que un objeto B colabora con otro objeto A, si el objeto A para cumplir con una responsabilidad, necesita enviar algún mensaje a B solicitándole un servicio. Desde el punto de vista del cliente, "A", cada una de sus colaboraciones están asociadas con una responsabilidad particular implementada por el servidor, "B".

Mensaje

Un mensaje enviado de un objeto a otro representa la existencia de un *enlace* entre ambos. Los mensajes se muestran como líneas dirigidas, que representan su dirección, con una etiqueta que nombra al propio mensaje. Aunque el paso de mensajes es iniciado por el cliente y dirigido hacia el servidor los datos pueden fluir en ambas direcciones a través de un enlace:

- del cliente al servidor: parámetros
- del servidor al cliente: respuesta

Supongamos que el objeto Caja de Ahorro, que conoce quién es su titular, como parte de su responsabilidad de mostrar sus datos completos debe mostrar el nombre del titular. Entonces necesita pedirle al objeto Titular que le retorne su nombre, para ello le enviará un mensaje indicándole tal solicitud. El objeto Titular responderá retornándole su nombre. Así se manifiesta el *enlace* entre ambos objetos. Esto puede representarse gráficamente de la siguiente manera:



El gráfico donde se muestran los enlaces entre objetos se denomina "Diagrama de Colaboraciones" y será objeto de estudio más adelante.

Relación de Agregación

La agregación denota una jerarquía todo/parte, en la cual un objeto del todo *tiene* objetos de la parte.

La agregación puede o no denotar contención física. Por ejemplo, un aeroplano se compone de alas, motores, tren de aterrizaje, etc.: es un caso de contención física. En otro ejemplo, un accionista tiene acciones, pero las acciones no son de ninguna manera parte física del accionista. Esta última relación todo/parte es más conceptual y por lo tanto menos directa que la agregación física de las partes que forman un aeroplano.

Se desarrolla este tema con más detalle en el punto siguiente, "Relaciones entre Clases".

Las clases, al igual que los objetos, no existen aisladamente. Antes bien, para un dominio de problema específico las abstracciones clave suelen estar relacionadas por vías muy diversas e interesantes, formando la estructura de clases.

Existen tres tipos básicos de relaciones entre clases, en UML:

- Generalización/Especialización, que denota una relación del tipo **"es un"**, **"padre/hijo"**, conocida como **herencia**.
- Asociación, que denota alguna dependencia semántica entre clases de otro modo independientes.
- Dependencia, es una relación de uso, se usarán cuando se quiera indicar que un elemento utiliza a otro.

Relación de generalización

La herencia es una implantación de la generalización. La generalización establece que las propiedades de un tipo se aplican a sus subtipos. La herencia hace que la estructura de datos y operaciones estén disponibles para su reutilización por parte de sus subclases. La herencia de las operaciones de una superclase permite que las clases compartan el código (en vez de volverlo a definir). La herencia de estructura de datos permite la reutilización de la estructura.

Dicho sencillamente, la herencia es una relación entre clases en la que una clase comparte la estructura y/o el comportamiento definidos en una (herencia simple) o más clases (herencia múltiple). La clase de la que otras heredan se denomina *superclase* o *clase padre* y la clase que hereda de otra o más clases se denomina *subclase* o *clase hija*.

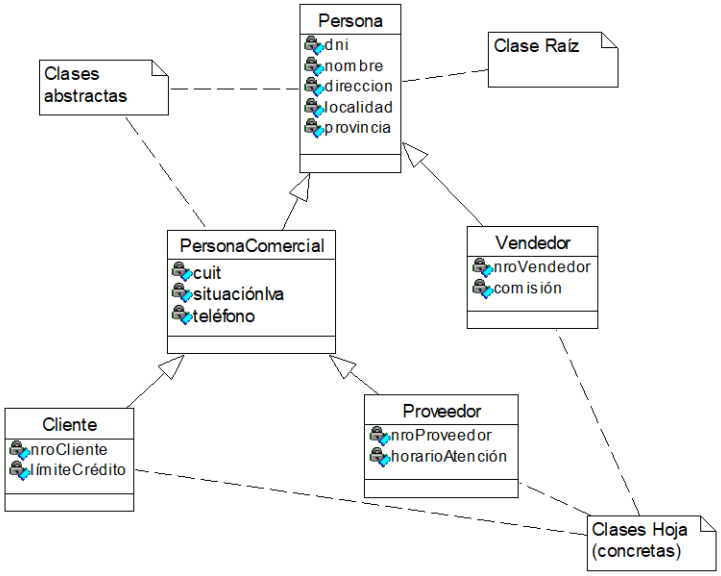
A menudo, una subclase añade atributos y operaciones a los que hereda de sus padres. Una operación de un hijo con la misma signatura (nombre, parámetros) que una operación del padre, redefine la operación heredada del padre; esto se conoce como *polimorfismo*, haciendo alusión a una operación que adopta varias formas de implantación según haya sido redefinida en las clases hijas.

Una clase puede tener ninguno, uno o más padres. Una clase sin padres y uno o más hijos se denomina *raíz* o *clase base*. Una clase sin hijos se denomina *clase hoja*. Una clase con un único padre se dice que utiliza herencia simple; una clase con más de un padre se dice que utiliza herencia múltiple.

Clases concretas y abstractas

Las clases hojas son de las que se espera que existan instancias, es decir objetos, por ello se las denomina también *clases concretas*. Las clases sin instancias se llaman *clases abstractas*. Una clase abstracta se redacta con la idea de que las subclases añadan cosas a su estructura y comportamiento, usualmente completando la implementación de sus métodos. De modo que nunca existirán objetos de una clase abstracta.

Gráficamente, la generalización se representa como una línea dirigida continua, con una punta de flecha vacía apuntando al padre, como se muestra en el siguiente ejemplo:

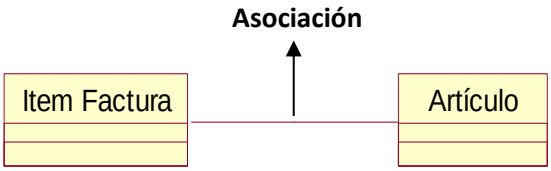


Relación de Asociación

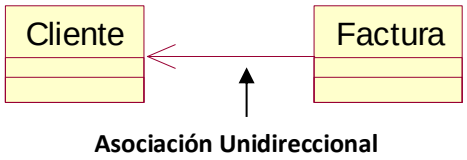
Una asociación es una relación estructural que especifica que los objetos de una clase están conectados con los objetos de otra. Una asociación sólo denota una dependencia semántica entre dos clases, pero no establece la forma exacta en que una clase se relaciona con la otra; sólo puede denotarse esa semántica nombrando el papel que desempeña cada clase en relación con la otra.

Dada una asociación entre dos clases se puede navegar desde un objeto de una clase hasta un objeto de la otra.

Gráficamente, una asociación se representa como una línea continua que conecta la misma o diferentes clases:



Un concepto importante para modelar respecto de una asociación es la **navegación**. La navegación de una asociación especifica que dado un objeto perteneciente a la clase de un extremo se puede llegar fácil y directamente a los objetos de la clase del otro extremo, normalmente debido a que el objeto inicial almacena algunas referencias a los objetos en el destino. La dirección de la **navegación** indica qué clase es la que contiene la referencia hacia la otra (determina "quién conoce a quién").

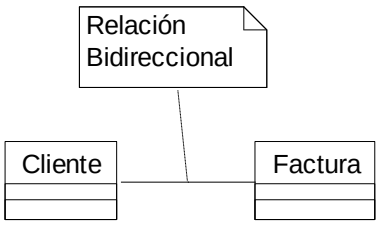


En este ejemplo Factura conoce a Cliente

Puede ocurrir que en algunos casos la asociación necesite ser navegable en ambos sentidos. Por ejemplo:

- Para mostrar los datos completos de una factura es necesario que factura conozca al cliente al que corresponde.
- Para mostrar un resumen de cuenta de un cliente es necesario que cliente conozca las facturas que le corresponden.

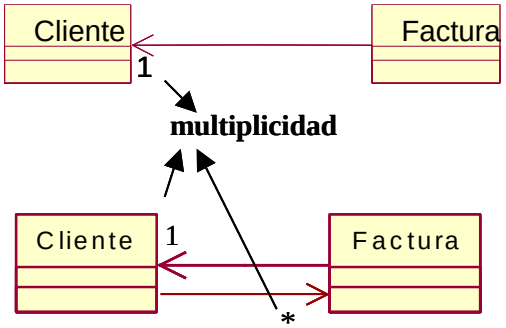
Esta situación se modela de la siguiente manera:



O también se puede representar así:

Multiplicidad

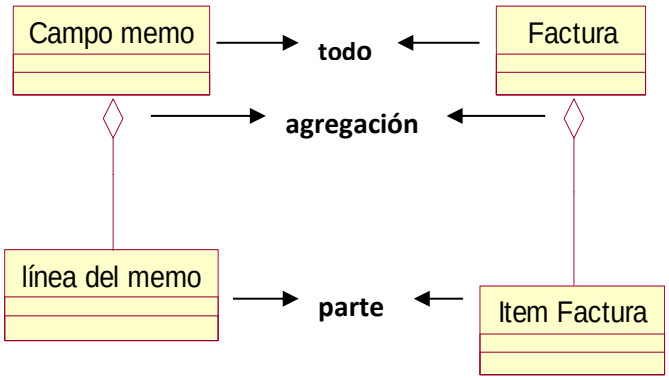
En muchas situaciones de modelado, es importante señalar cuántos objetos pueden conectarse a través de una instancia de la asociación. Este "cuántos" se denomina **multiplicidad** de la asociación y se escribe como una expresión que se evalúa a un rango de valores o a un valor explícito. Cuando se indica una multiplicidad en un extremo de una asociación se está especificando que para cada objeto de la clase en el extremo opuesto debe haber tantos objetos en este extremo. Se puede indicar una multiplicidad de exactamente uno (1), cero (0), muchos (*), uno o más (1..*) o un valor exacto (por ejemplo, 3). La multiplicidad se indica en el extremo que corresponde a la navegación.



Relación de agregación

Un tipo especial de asociación, lo constituye la agregación. Las relaciones de agregación entre clases tienen un paralelismo directo con las relaciones de agregación entre los objetos correspondientes a esas clases. La agregación es una relación "todo/parte" en la cual una clase representa una cosa grande (el "todo") , que consta de elementos más pequeños (las "partes"). Representa una relación del tipo "tiene un", o sea, un objeto del todo tiene objetos de la parte.

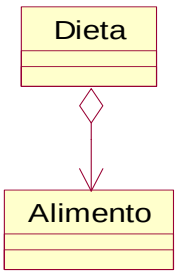
Gráficamente la agregación se representa como se muestra a continuación:



Como ya se comentó la agregación puede o no denotar contención física. Tenemos entonces dos posibilidades en cuanto a la relación de agregación:

- **Por Valor**: Es un tipo de relación, en donde el tiempo de vida del objeto incluido está condicionado por el tiempo de vida del que lo incluye. Este tipo de relación es también llamada **Composición** (el Objeto base se contruye a partir del objeto incluido, es decir, es "parte/todo"). En este tipo de relación el objeto "parte" no existe independientemente del objeto "todo" que lo contiene. Esto significa que el tiempo de vida de ambos objetos está íntimamente relacionado: cuando se crea una instancia del todo, se crea también por lo menos una instancia de la parte; cuando se destruye el objeto "todo" esto implica la destrucción de todos los objetos "parte" relacionados a él. En el ejemplo, cuando se crea una instancia de factura (un objeto factura) es porque existe al menos una instancia (objeto) item factura.

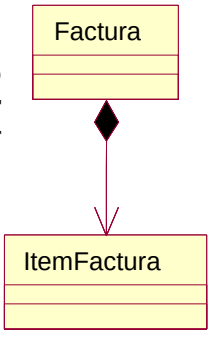
- **Por Referencia**: Es un tipo de relación, en donde el tiempo de vida del objeto incluido es independiente del que lo incluye. Este tipo de relación es comúnmente llamada **Agregación** (el objeto base utiliza al incluido para su funcionamiento). Por ejemplo:

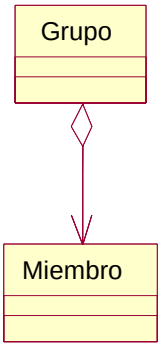


Una determinada dieta puede dejar de existir, pero ello no implica que ocurra lo mismo con los alimentos que ella contenía. Estos pueden seguir existiendo y estar contenidos en otras dietas.

Si se desea mostrar gráficamente el tipo de agregación al que nos estamos refiriendo, se puede proceder de la siguiente manera:

La composición o agregación **por Valor** se destaca por un rombo relleno





La agregación **por Referencia** se destaca por un rombo transparente

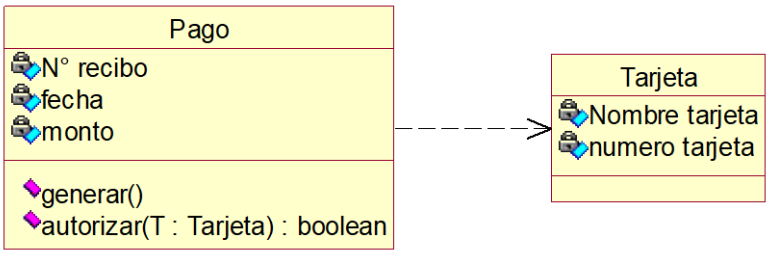
Esta forma de graficar es opcional, no es obligatorio determinar el tipo de agregación, pero es útil comprender el significado de ambas.

Relación de Dependencia

Este tipo de relación se define como "una relación de uso que declara que un cambio en la especificación de un elemento puede afectar a otro elemento que la utiliza", esto representa la dependencia que tiene una clase de otra.

Generalmente las dependencias se utilizan para indicar que una clase utiliza a otra como argumento en la signatura de una operación. Esto es claramente una relación de uso: si la clase utilizada cambia, la operación de la otra clase puede verse también afectada, porque la clase utilizada puede presentar ahora una interfaz o comportamientos diferentes.

Gráficamente estas relaciones se representan con una línea discontinua dirigida hacia la clase de la cual se depende, como se muestra a continuación:



La dependencia se utiliza para representar la visibilidad de parámetro. En nuestro ejemplo Tarjeta se hace visible para Pago porque un objeto de esta última clase recibirá como parámetro de entrada para su operación "autorizar", un objeto de la clase Tarjeta.

Modelando casos de uso del dominio del problema

Diagrama de caso de uso

Los casos de uso proporcionan un medio intuitivo y sistemático para capturar los requisitos funcionales poniendo énfasis en el valor añadido para cada usuario individual o para cada sistema externo. La utilización de los casos de uso hace que los analistas deben pensar en términos de quiénes son los usuarios y qué necesitad u objetivos de la empresa pueden cumplir.

El principal esfuerzo de la fase de requisitos es desarrollar un modelo del sistema que se va a construir y la utilización de los casos de uso es una forma adecuada para ello, debido a que los requisitos funcionales se estructuran naturalmente mediante los casos de uso y los requisitos no funcionales suelen ser específicos de un caso de uso y pueden tratarse en ese contexto.

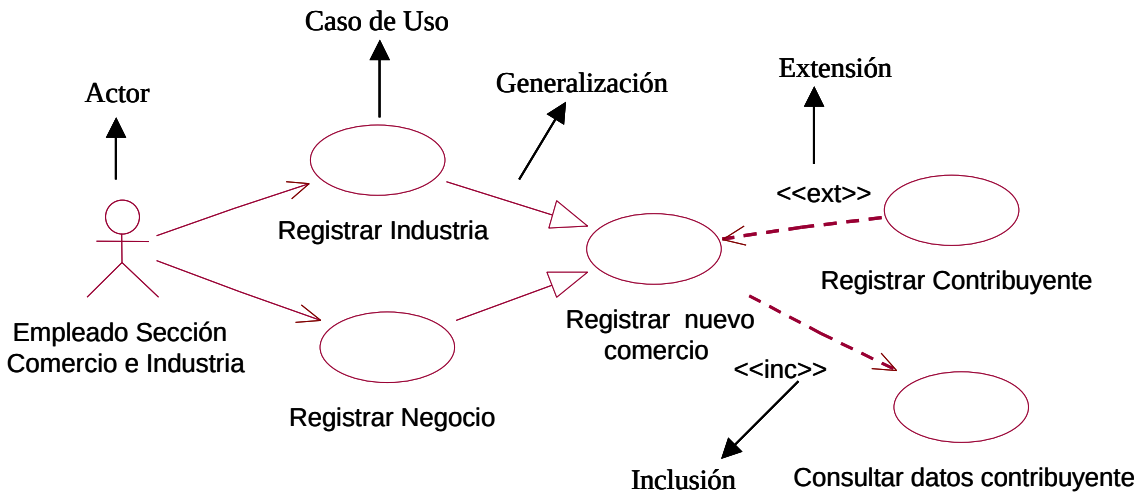
El modelo de casos de uso permite que los desarrolladores y los clientes lleguen a un acuerdo sobre los requisitos, es decir sobre lo que debe cumplir el sistema y constituye la entrada principal para el análisis, el diseño y las pruebas.

Los diagramas de casos de uso son importantes para modelar el comportamiento de un sistema o un subsistema. Estos diagramas facilitan que los sistemas y subsistemas sean abordables y comprensibles, al presentar una vista externa de cómo pueden utilizarse estos elementos en un contexto dado.

Los diagramas de casos de uso contienen: casos de uso, actores, relaciones de dependencia, generalización y asociación, tal que los actores se conectan a los casos de uso a través de asociaciones.

Una asociación entre un actor y un caso de uso indica que el actor y el caso de uso se comunican entre sí y cada uno puede enviar y recibir mensajes.

Simbología del Diagrama de Casos de Uso



El modelo de casos de uso es un modelo que contiene actores, casos de uso y las relaciones entre éstos.

Actor

Un actor es el rol que juega un usuario en un caso de uso. Normalmente, un actor representa un rol que es jugado por una persona, un dispositivo de hardware o incluso otro sistema al interactuar con nuestro sistema, es un usuario del sistema. El modelo de casos de uso describe lo que hace el sistema para cada tipo de usuario. Cada usuario puede representarse por uno o más actores. De la misma forma cada sistema externo que interactúa con el sistema en desarrollo puede representarse por uno o más actores.

Cada vez que un usuario en concreto (un humano u otro sistema) interactúa con el sistema, la instancia correspondiente del actor está desarrollando ese papel. Una instancia de un actor es, por lo tanto, un usuario concreto que interactúa con el sistema.

Nombre de actor

El nombre del actor debe reflejar el rol que cumple un usuario al interactuar con el caso de uso al que está conectado.

Caso de Uso

Un caso uso representa cada forma en que los actores usan el sistema. Los casos de uso son fragmentos de funcionalidad que el sistema ofrece para aportar un resultado de valor para sus actores.

Nombre de caso de uso

Cada caso de uso debe tener un NUMERO y un NOMBRE que lo distinga de los demás. Puede ser un nombre simple (una simple cadena de texto) o un nombre de camino en el caso de que esté precedido por el nombre del paquete en que está incluido el caso de uso. Los nombres de casos de uso son expresiones verbales que describen algún comportamiento del vocabulario del sistema que se está modelando.

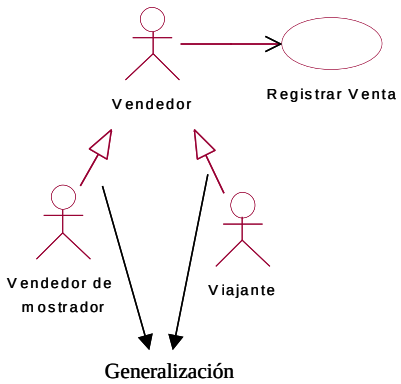
Nro + Verbo en infinitivo + (objeto del dominio del problema)

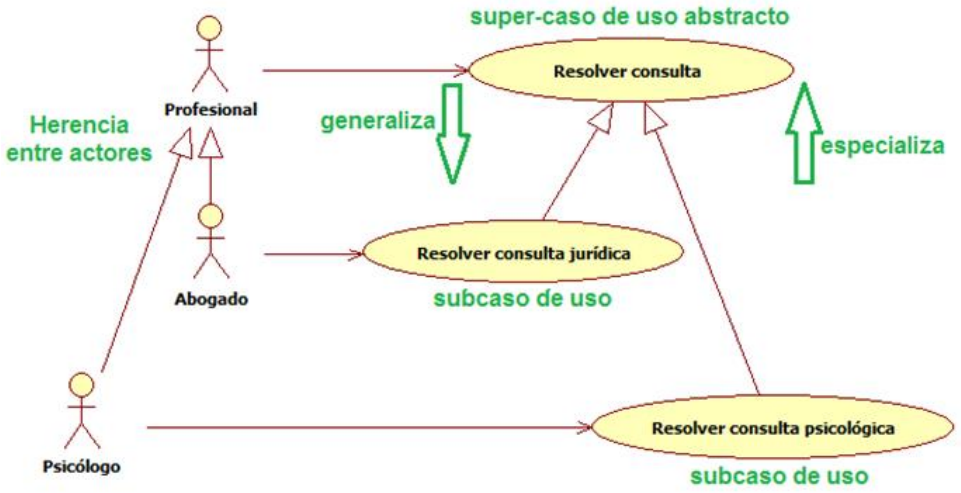
Un caso de uso especifica una secuencia de acciones que el sistema puede llevar a cabo interactuando con sus actores, incluyendo secuencias dentro de la secuencia.

Relación	Símbolo	Significado
Comunica	———	Para conectar un actor con un caso de uso se utiliza una línea sin puntas de flecha.
Incluye	<< Incluye >> <----------	Un caso de uso contiene un comportamiento común para más de un caso de uso. La flecha apunta al caso de uso común.
Extiende	<< Extiende >> ---------->	Un caso de uso distinto maneja las excepciones del caso de uso básico. La flecha apunta del caso de uso extendido al básico.
Generaliza	———▷	Una "cosa" de UML es más general que otra "cosa". La flecha apunta a la "cosa" general.

Relación de Generalización entre actores

Pueden definirse categorías generales de actores y especializarlos a través de relaciones de generalización. Los actores especializados (hijos) heredan el comportamiento del actor padre. Si un caso de uso es instanciado por el actor "padre" puede ser instanciado por cualquiera de sus hijos. Ahora bien, podría haber casos de uso que son instanciados por uno de los actores "hijo" en particular.





Relación de generalización/inclusión y extensión entre casos de uso

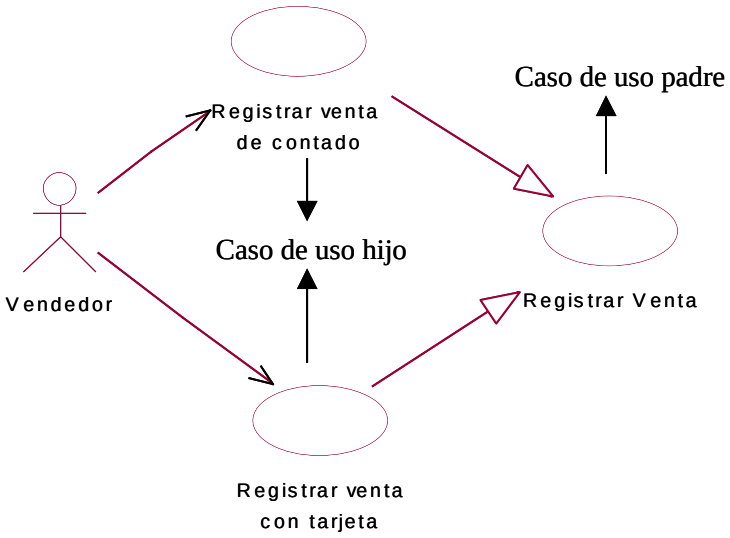
Los casos de uso pueden organizarse especificando relaciones de generalización, inclusión y extensión entre ellos. Estas relaciones se utilizan para factorizar el comportamiento común (extrayendo ese comportamiento de los casos de uso en los que se incluye) y para factorizar variantes (poniendo ese comportamiento en otros casos de uso que lo extienden).

Generalización

La generalización entre casos de uso es como la generalización entre clases. En este contexto significa que el caso de uso hijo hereda el comportamiento del caso de uso padre. El hijo puede modificar y/o agregar comportamiento al heredado.

La generalización se emplea para simplificar la forma de trabajo y la comprensión del modelo de casos de uso y para reutilizar casos de uso "semifabricados". Un caso de uso "semifabricado" existe solamente para que otros lo reutilicen.

Gráficamente se indica, al igual que la herencia entre clases, con una línea continua con punta de flecha vacía dirigida del caso de uso hijo hacia el padre.

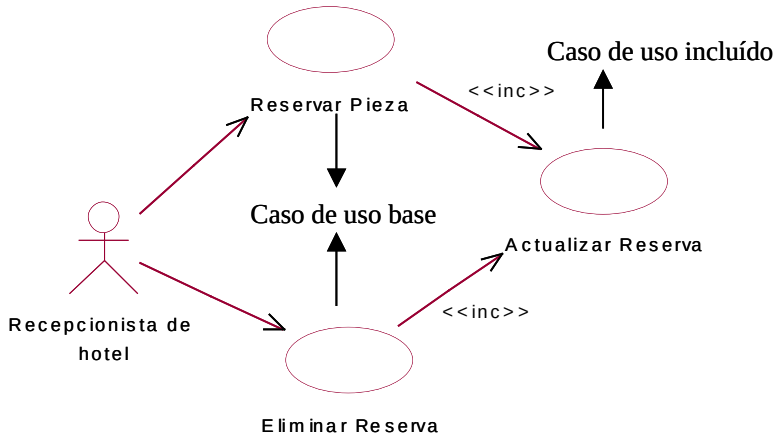


Inclusión

Una relación de inclusión entre casos de uso significa que un caso de uso base incorpora explícitamente el comportamiento de otro caso de uso en el lugar especificado en el caso base. El caso de uso incluido nunca es instanciado por un actor, sino que es instanciado por el/los casos de uso que lo incluyen. Por ello, un caso de uso de inclusión es siempre un caso de uso abstracto.

La relación de inclusión se usa para abstraer el comportamiento común entre casos de uso, evitando describir el mismo flujo de eventos repetidas veces. La secuencia de comportamiento y los atributos del caso de uso incluido se encapsulan y no pueden modificarse o accederse, solamente puede utilizarse el resultado (o función) del caso de uso incluido.

La inclusión se representa como una dependencia estereotipada con la palabra include o en forma abreviada inc, dirigida desde el caso de uso base hacia el caso de uso incluido.

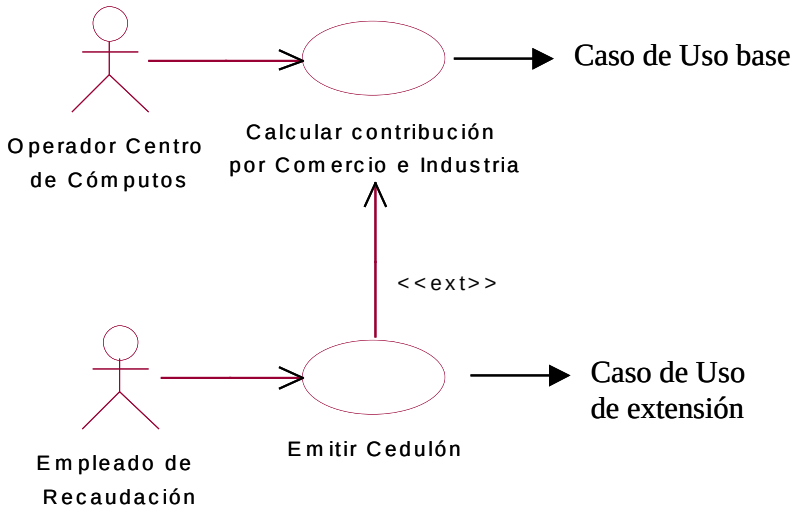


Extensión

Una relación de extensión entre casos de uso significa que un caso de uso base incorpora implícitamente el comportamiento de otro caso de uso. El caso de uso base puede ejecutarse aisladamente, pero bajo ciertas condiciones su funcionalidad será extendida por la del caso de uso de extensión.

La extensión puede verse como que el caso de uso que extiende incorpora su comportamiento en el caso de uso base. Una relación de extensión se utiliza para modelar la parte de un caso de uso que el usuario puede ver como comportamiento opcional del sistema. De esta forma se separa el comportamiento opcional del obligatorio. También puede utilizarse para modelar un subflujo separado que sólo se ejecuta bajo ciertas circunstancias.

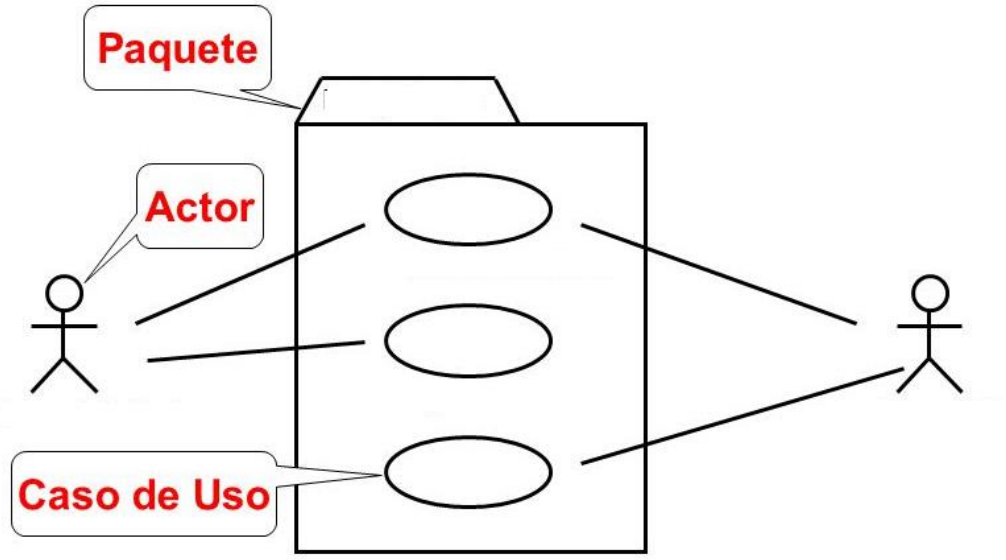
El caso de uso de extensión puede en algunos casos ser instanciado directamente por un actor (en este caso se considera un caso de uso *concreto*), además de instanciarse para extender el comportamiento de un caso de uso base. Si el caso de uso de extensión nunca es instanciado directamente por un actor, entonces es un caso de uso *abstracto*.



La extensión se representa como una dependencia estereotipada con la palabra *extend* o en forma abreviada *ext*, dirigida desde el caso de uso de extensión hacia el caso de uso base.

Si el modelo de casos de uso es grande, es decir si el número de ellos es elevado es útil introducir el concepto de “Paquete” y representarlo con UML en el modelo para tratar su tamaño.

Un paquete reunirá cierto número de casos de uso agrupados por algún criterio de homogeneidad: los que corresponden a un mismo actor, los que se refieren a un mismo proceso, etc.

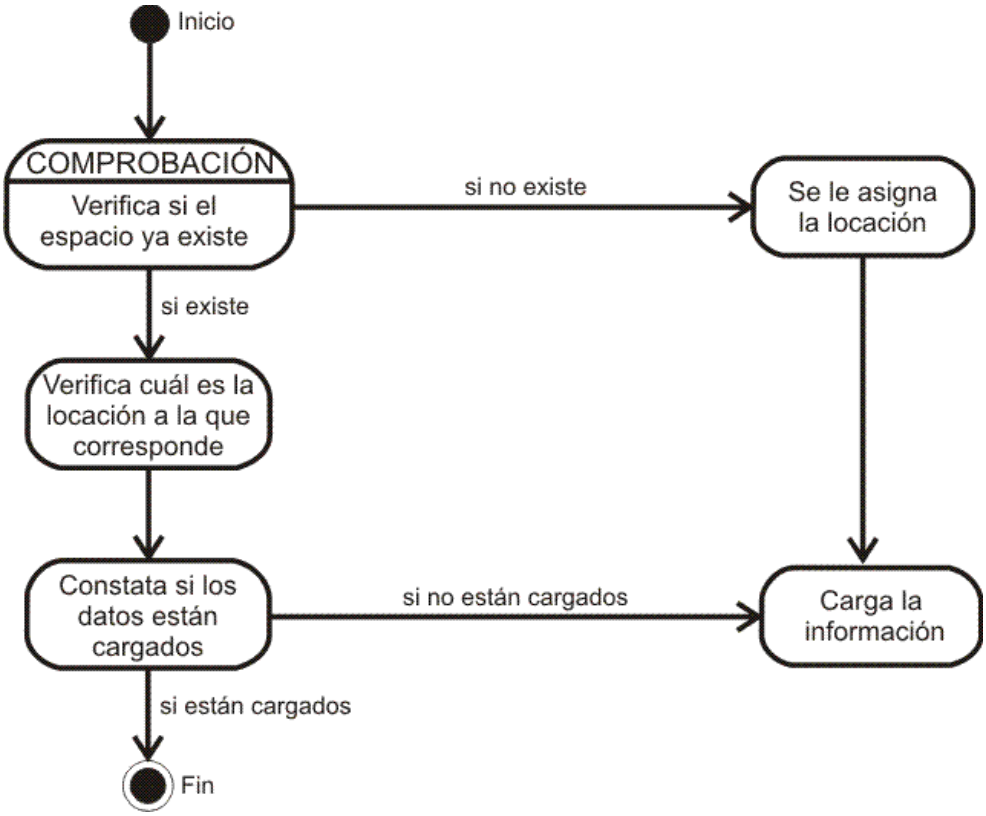


A pesar de ser considerada una técnica de Análisis Orientado a Objetos, es importante destacar que los casos de uso poco tienen que ver con entender a un sistema como un conjunto de objetos que interactúan, que es la premisa básica del análisis orientado a objetos "clásico". En este sentido, el éxito de los casos de uso no hace más que dar la razón al análisis estructurado, que propone que la mejor forma de empezar a entender un sistema es a partir de los servicios o funciones que ofrece a su entorno, independientemente de los objetos que interactúan dentro del sistema para proveerlos.

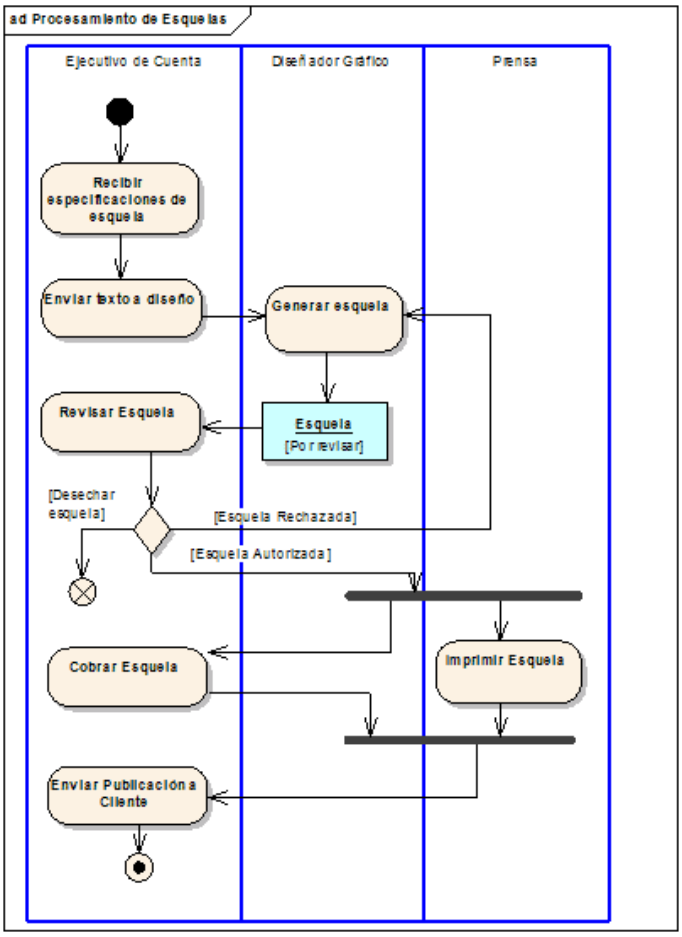
Por lo expresado, un caso de uso es una "especificación". Especifica el comportamiento de "cosas" dinámicas, en este caso instancias de casos de uso. Una instancia de un caso de uso es la realización (o ejecución) de un caso de uso.

Una descripción de un caso de uso puede incluir:

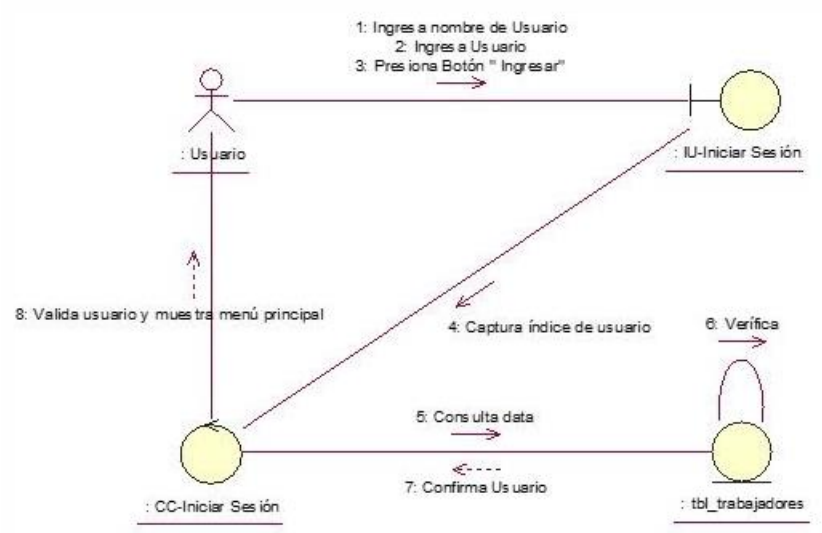
- ***Diagramas de estados***: Especifican el ciclo de vida de las instancias de casos de uso.



- ***Diagramas de actividad***: Describe el ciclo de vida con más detalle indicando la secuencia temporal de acciones que tienen lugar dentro de cada transición.



- ***Diagramas de colaboración o COMUNICACION y diagrama de secuencia***: Describen las interacciones entre una instancia típica de un actor y una instancia típica de un caso de uso.

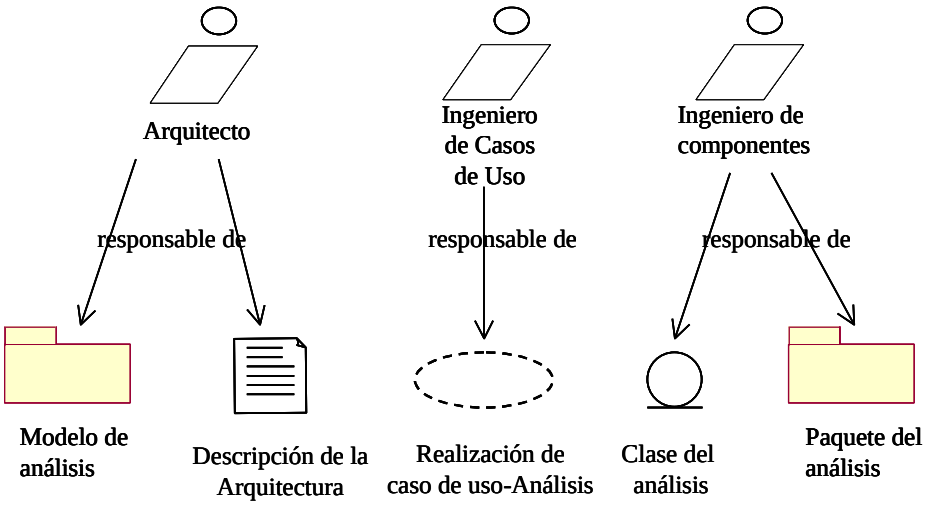


Flujo de Trabajo de Análisis

Para describir el flujo de trabajo de análisis enunciaremos, y luego detallaremos, los artefactos creados en este flujo de trabajo, los trabajadores participantes y las actividades a realizar:

- **Artefactos**: Los artefactos fundamentales que se utilizan en análisis son:
  - Modelo de análisis
  - Clase de análisis, que comprende las clases de interfaz, entidad y control.
  - Realización de caso de uso – análisis, que implica principalmente el diagrama de colaboraciones.
  - Paquete del análisis.
  - Descripción de la arquitectura (vista del modelo de análisis).
- **Trabajadores**: Los trabajadores responsables por los artefactos que se producen en el modelado de análisis:
  - Arquitecto
  - Ingeniero de casos de uso
  - Ingeniero de componentes
- **Actividades:** Las actividades a realizar por los trabajadores para producir los distintos artefactos son:
  - Analizar la arquitectura.
  - Analizar un caso de uso.
  - Analizar una clase.
  - Analizar un paquete.

El siguiente gráfico muestra la relación entre los artefactos del análisis y los trabajadores responsables de cada uno:



A continuación, se muestra un gráfico que indica el flujo de trabajo para la actividad de análisis, que relaciona los trabajadores participantes con sus actividades y pone de manifiesto la secuencia de éstas:

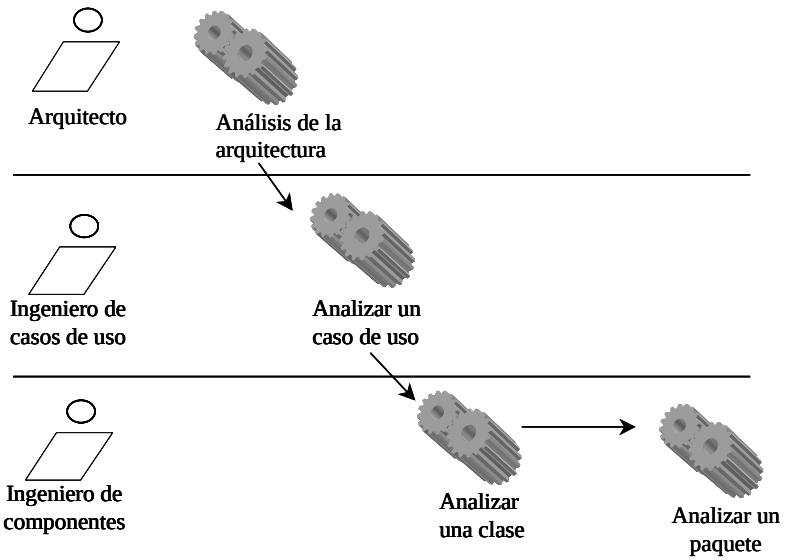


Diagrama de Interacción

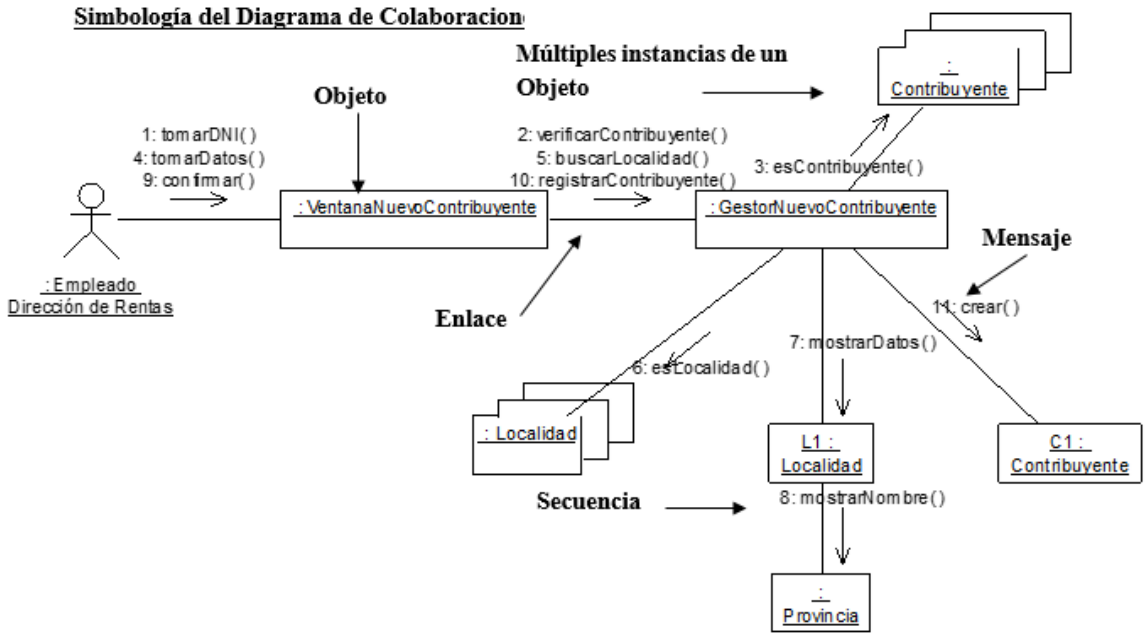
La secuencia de acciones en un CASO DE USO comienza cuando un actor invoca el caso de uso mediante el envío de un mensaje al sistema. Entonces, un objeto de interfaz recibirá ese mensaje, a su vez, enviará un mensaje a algún otro objeto y de esta forma los objetos interactúan y se comunican para llevar a cabo el caso de uso.

Hay dos tipos de diagramas de interacción: el diagrama de colaboración y el diagrama de secuencia. En análisis es preferible utilizar el diagrama de colaboración, ya que aquí el objetivo fundamental es identificar responsabilidades de los objetos y no tiene tanta importancia la secuencia detallada y ordenada cronológicamente. Por lo tanto, dejaremos los diagramas de secuencia para la actividad de diseño.

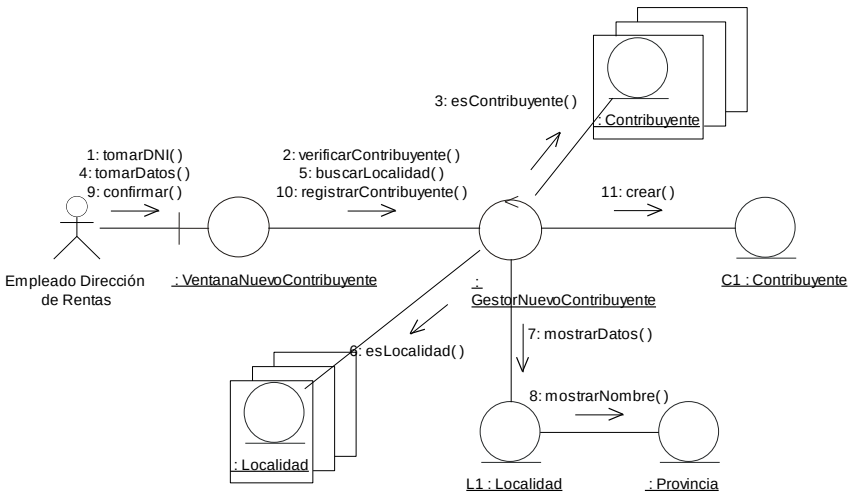
Diagrama de colaboración

Un *diagrama de colaboración* o COMUNICACIÓN destaca la organización de los objetos que participan en una interacción o "REALIZACION" DE UN CASO DE USO.

Este diagrama se construye colocando en primer lugar los objetos que participan en la colaboración como nodos del grafo. A continuación, se representan los enlaces que conectan esos objetos como arcos del grafo. Por último, esos enlaces se "adornan" con los mensajes que envían y reciben los objetos.

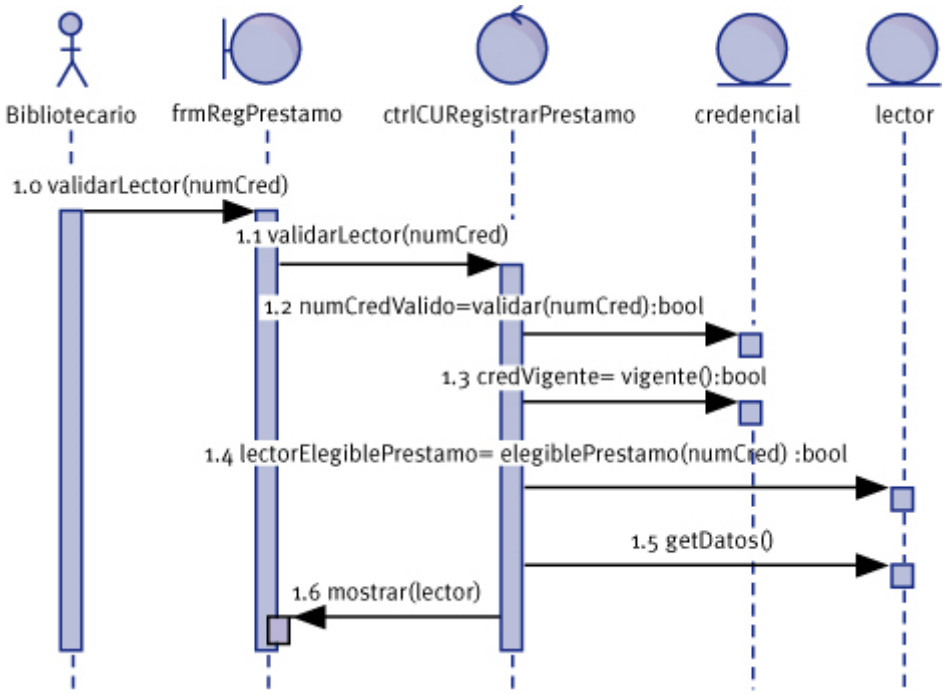


En este caso no se graficaron los objetos de las clases de análisis utilizando los estereotipos, sino el símbolo de objeto común. Ambos son equivalentes. Utilizando los estereotipos un diagrama de colaboración se vería de la siguiente manera:



- Una *interacción* es un comportamiento que comprende un conjunto de mensajes intercambiados entre un conjunto de objetos dentro de un contexto para lograr un propósito.
- Un *mensaje* es la especificación de una comunicación entre objetos que transmite información, a la espera de que se desencadene una actividad.
- Un *enlace* es una conexión semántica entre dos objetos. Es una instancia de una asociación.

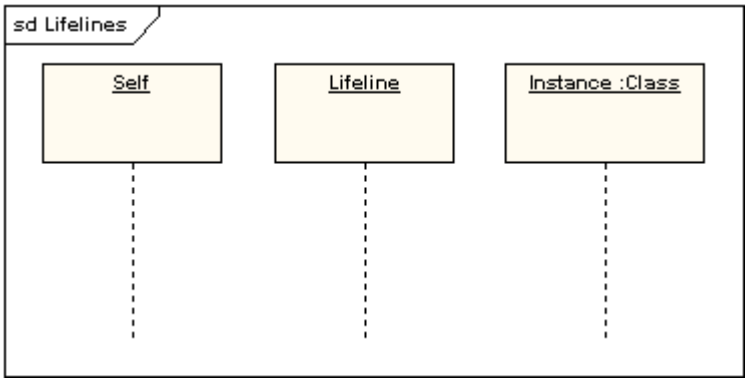
Diagrama de secuencia



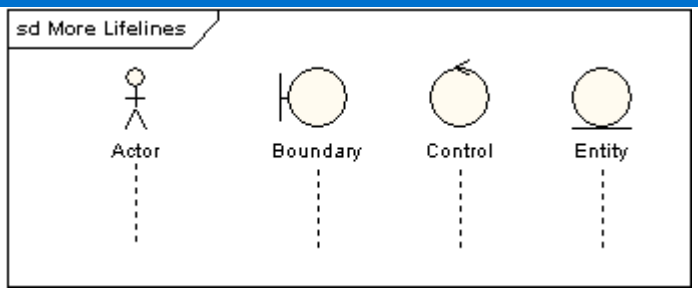
Un diagrama de secuencia es una forma de diagrama de interacción que muestra los objetos como líneas de vida a lo largo de la página y con sus interacciones en el tiempo representadas como mensajes dibujados como flechas desde la línea de vida origen hasta la línea de vida destino. Los diagramas de secuencia son buenos para mostrar qué objetos se comunican con qué otros objetos y qué mensajes disparan esas comunicaciones. Los diagramas de secuencia no están pensados para mostrar lógicas de procedimientos complejos.

Línea de Vida

Una línea de vida representa un participante individual en un diagrama de secuencia. Una línea de vida usualmente tiene un rectángulo que contiene el nombre del objeto. Si el nombre es self entonces eso indica que la línea de vida representa el clasificador que posee el diagrama de secuencia.

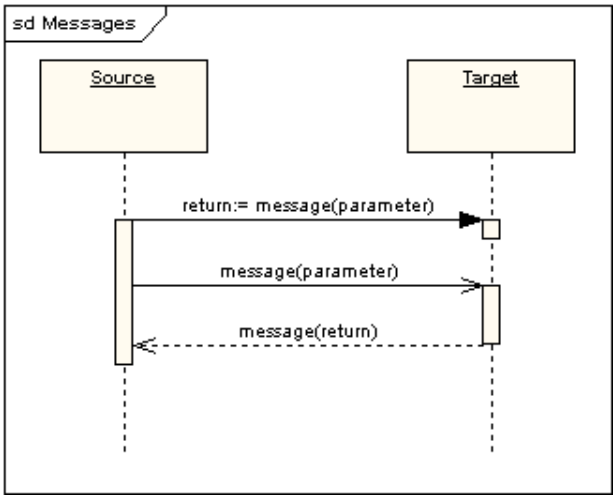


Algunas veces un diagrama de secuencia tendrá una línea de vida con un símbolo del elemento actor en la parte superior. Este usualmente sería el caso si un diagrama de secuencia es contenido por un caso de uso. Los elementos entidad, control y límite de los diagramas de robustez también pueden contener líneas de vida.



Mensajes

Los mensajes se muestran como flechas. Los mensajes pueden ser completos, perdidos o encontrados; síncronos o asíncronos: llamadas o señales. En el siguiente diagrama, el primer mensaje es un mensaje síncrono (denotado por una punta de flecha oscura), completo con un mensaje de retorno implícito; el segundo mensaje es asíncrono (denotado por una punta de flecha en línea) y el tercero es un mensaje de retorno asíncrono (denotado por una línea punteada).

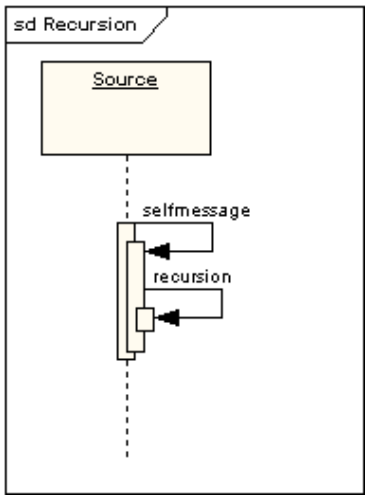


Ocurrencia de ejecución

Un rectángulo fino a lo largo de la línea de vida denota la ocurrencia de ejecución o activación de un foco de control. En el diagrama anterior hay tres ocurrencias de ejecución. El primero es el objeto origen que envía dos mensajes y recibe dos respuestas, el segundo es el objeto destino que recibe un mensaje asíncrono y retorna una respuesta, y el tercero es el objeto destino que recibe un mensaje asíncrono y retorna una respuesta.

Mensaje Self

Un mensaje self puede representar una llamada recursiva de una operación, o un método llamando a otro método perteneciente al mismo objeto. Este se muestra como cuando crea un foco de control anidado en la ocurrencia de ejecución de la línea de vida.

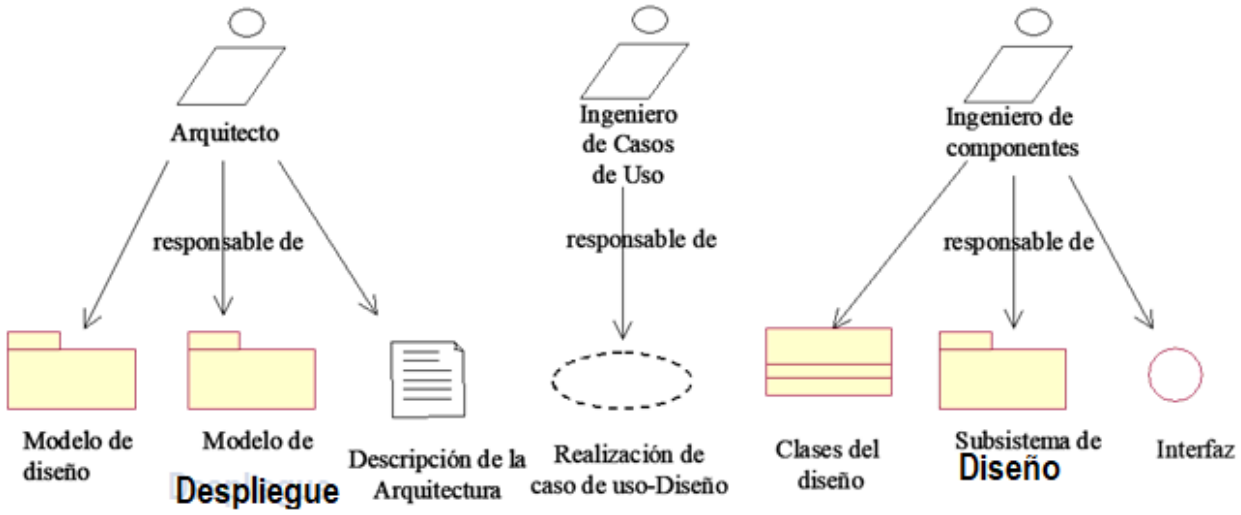


Flujo de Trabajo de Diseño

Entre los propósitos del diseño se puede destacar:

- Comprender en profundidad los aspectos relacionados con los REQUISITOS NO FUNCIONALES y RESTRICCIONES relacionadas con los lenguajes de programación, componentes reutilizables, sistemas operativos, tecnologías de distribución y concurrencia, tecnologías de interfaz de usuario, tecnologías de gestión de transacciones, etc.
- Producir una ENTRADA apropiada y un punto de partida para las actividades de implementación.
- DESCOMPONER los trabajos de implementación en partes más manejables que puedan ser llevadas a cabo por diferentes equipos de desarrollo, teniendo en cuenta la concurrencia.
- Identificar las INTERFACES entre los subsistemas.
- Crear una ABSTRACCION sin costuras de la implementación del sistema, en el sentido de que la implementación es un refinamiento del diseño que rellena lo existente sin modificar la estructura. Esto permite la utilización de tecnologías como la generación de código.

El siguiente gráfico muestra la relación entre los artefactos del diseño y los trabajadores responsables de cada uno:



A continuación, se muestra un gráfico que indica el flujo de trabajo para la actividad de diseño, que relaciona los trabajadores participantes con sus actividades, poniendo de manifiesto la secuencia de éstas:

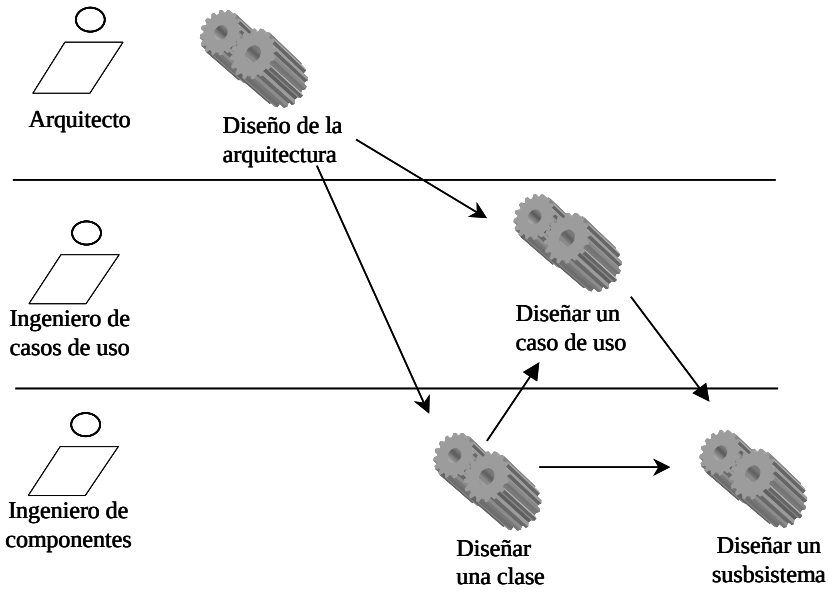
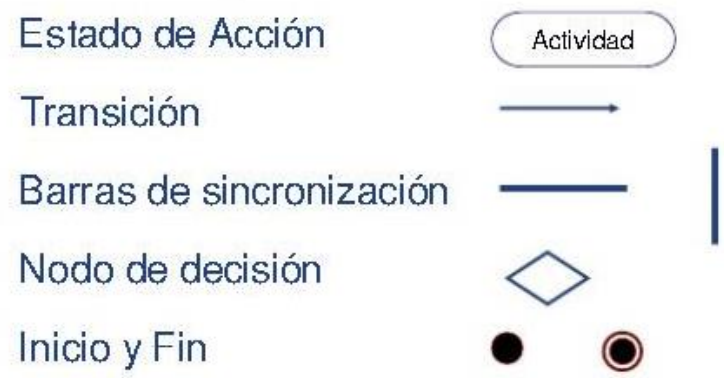
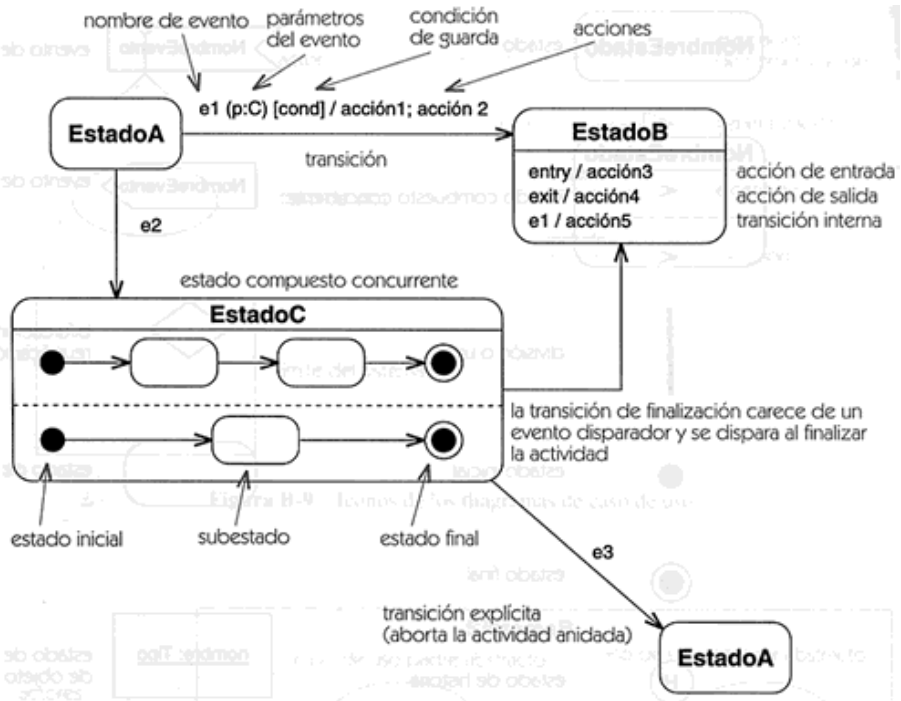


Diagrama de Estado

Los diagramas de estado son una técnica conocida para describir el comportamiento de un sistema. Describen todos los estados posibles en los que puede entrar un objeto particular y la manera en que cambia el estado del objeto, como resultado de los eventos que llegan a él. En la mayor parte de las técnicas Orientadas a Objetos, los diagramas de estado se dibujan para una sola clase, mostrando el comportamiento de un solo objeto durante todo su ciclo de vida.



Ejemplo:

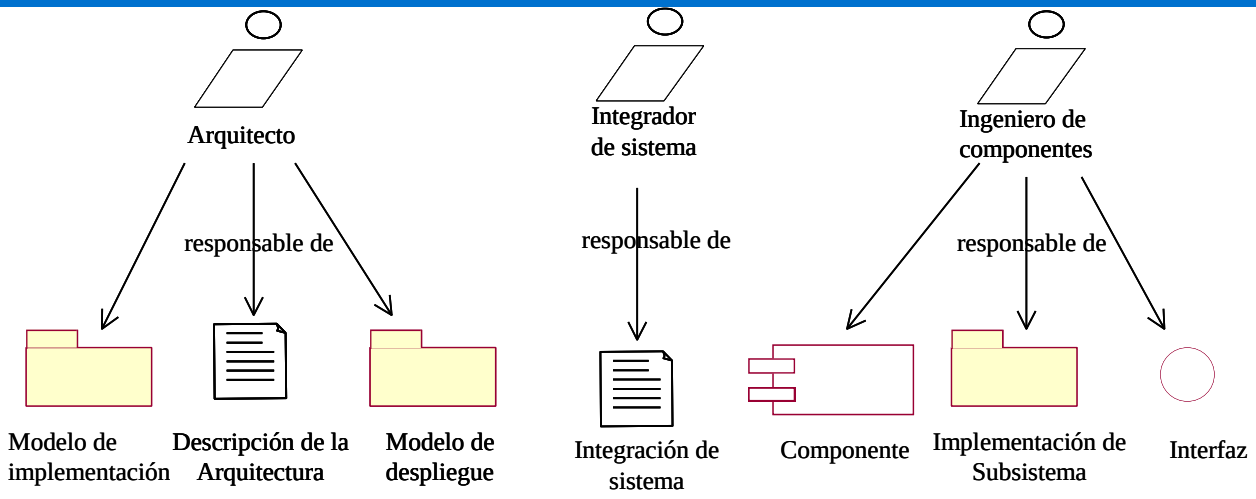


Flujo de Trabajo de Implementación

Entre los objetivos que se persiguen durante la implementación se puede mencionar:

1. Implementar las clases y subsistemas de diseño.
2. Probar los componentes individualmente, para luego integrarlos, compilarlos y enlazarlos en uno o más ejecutables que serán enviados para ser integrados y realizar las comprobaciones de sistema.
3. Asignar componentes ejecutables a nodos en el diagrama de despliegue, realizando así la distribución de la funcionalidad del sistema. Esto se basa fundamentalmente en las clases activas encontradas en el diseño.
4. Planificar las integraciones de sistemas necesarias en cada iteración. Como se sigue un enfoque incremental se obtiene un sistema que se implementa como una sucesión de pequeños pasos.

El siguiente gráfico muestra la relación entre los artefactos de la implementación y los trabajadores responsables de cada uno:



A continuación, se muestra un gráfico que indica el flujo de trabajo para la actividad de implementación, que relaciona los trabajadores participantes con sus actividades, poniendo de manifiesto la secuencia de éstas:

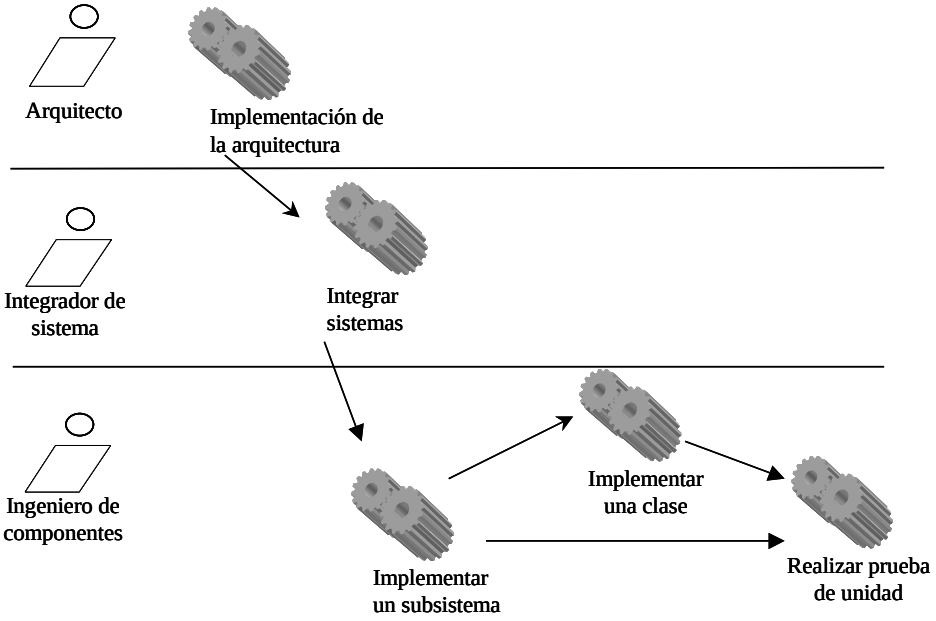
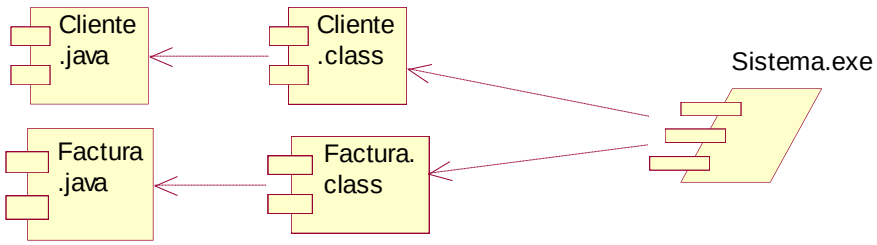


Diagrama de componentes

Un diagrama de componentes modela un conjunto de componentes y sus relaciones. Los diagramas de componentes contienen: componentes, interfaces y relaciones (dependencia, generalización, asociación, realización). Veamos un ejemplo de un fragmento de diagrama de componentes, en el que se indica qué componentes ejecutables implementan ciertos links de una página Web (otro estereotipo de componente)

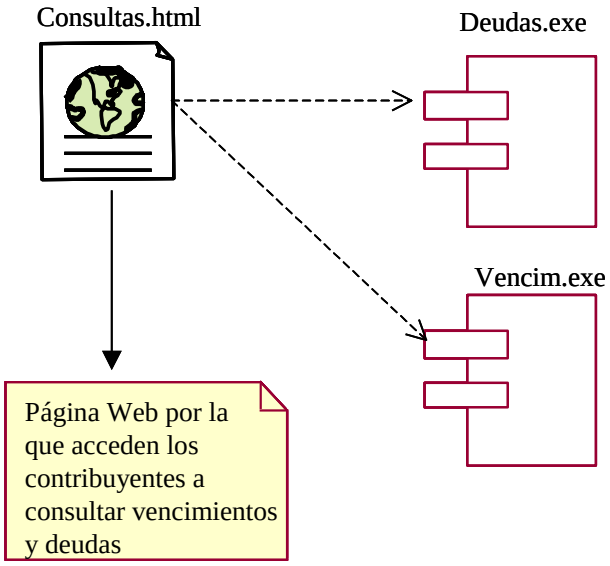
Otra posible forma de utilizar el diagrama de componentes sería, por ejemplo, para determinar cómo varios componentes –código fuente se empaquetan en un componente – ejecutable:



Flujo de Trabajo de Prueba

Durante este flujo de trabajo se procederá a verificar el resultado de la implementación probando cada construcción, tanto las intermedias como las versiones finales del sistema. Es importante tener en cuenta que las pruebas de software son:

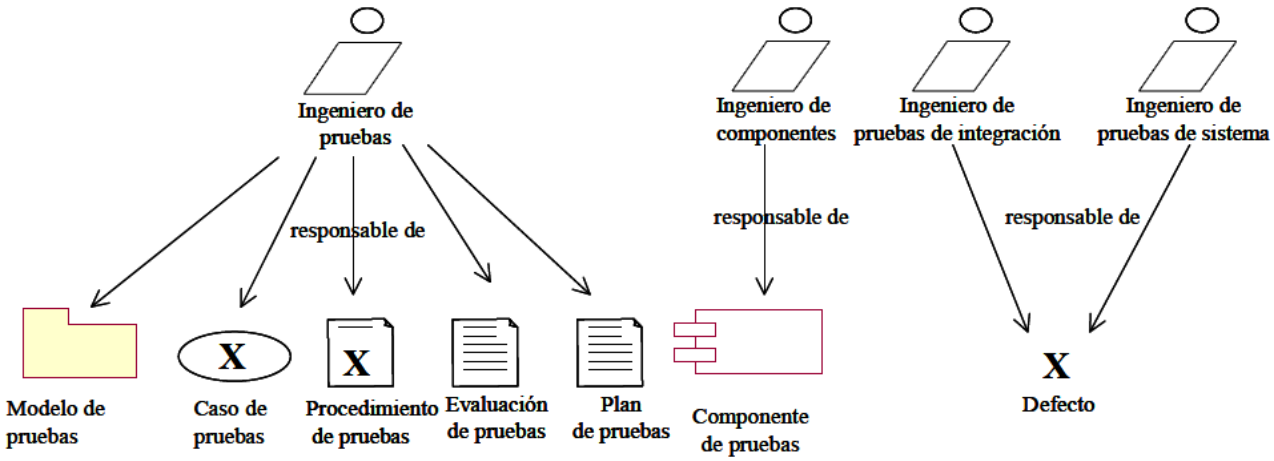
- Son un elemento crítico para la garantía de calidad de software y representa una revisión final de las especificaciones, diseño y codificación.
- Son siempre necesarias.
- En algunos casos ocupan un 40% del tiempo de un proyecto de sistemas.
- Las pruebas pretenden descubrir ERRORES.



Durante la fase de inicio puede hacerse algo de la planificación de las pruebas. Sin embargo, las pruebas se llevan a cabo cuando una construcción es sometida a las pruebas de integración y de sistema. Esto quiere decir que las actividades de pruebas se centran en las fases de elaboración cuando se prueba la línea base ejecutable del sistema, y en la fase de construcción cuando el grueso del sistema está implementado.

Durante la fase de transición el centro de atención se desplaza hacia la corrección de defectos durante los primeros usos del sistema y a las pruebas de regresión Flujo de trabajo de prueba.

El siguiente gráfico muestra la relación entre los artefactos de la Prueba y los trabajadores responsables de cada uno:



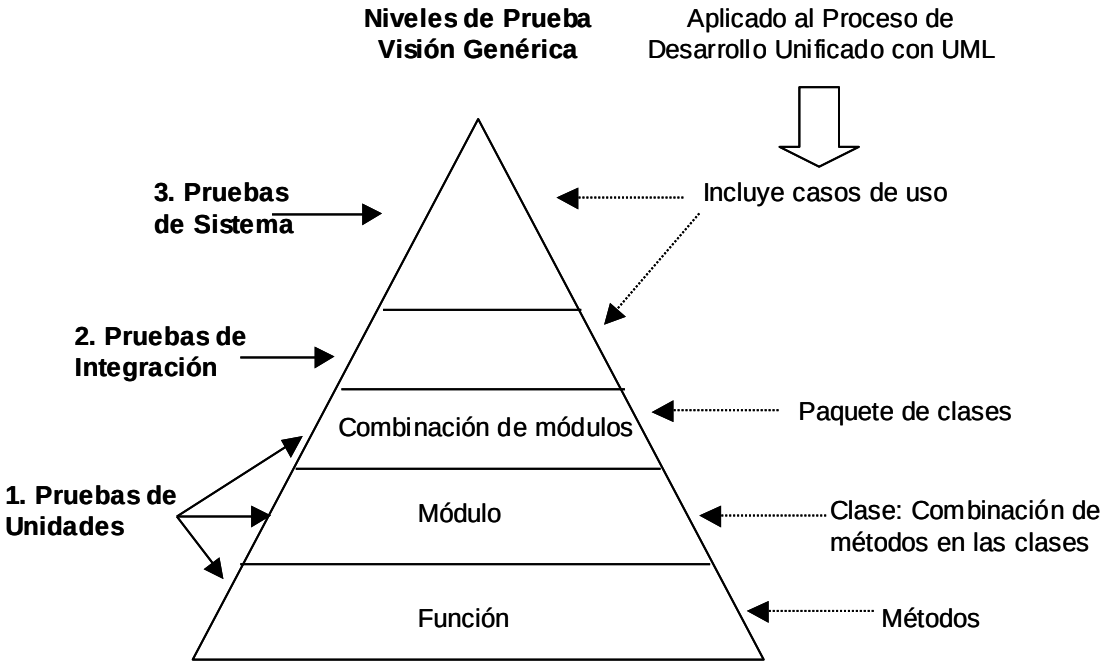
A continuación se muestra un gráfico que indica el flujo de trabajo para la actividad de Prueba, que relaciona los trabajadores participantes con sus actividades, poniendo de manifiesto la secuencia de éstas:

Flujo de Trabajo de Prueba

Niveles de Prueba

Todo proceso de prueba de software consiste en tres niveles de prueba: pruebas de unidad, de integración y de sistema. En el Proceso de Desarrollo Unificado, las pruebas de unidad se realizan el Flujo de Trabajo de Implementación y las pruebas de integración y de sistema en el Flujo de Trabajo de Prueba, que tratamos en esta unidad.

La siguiente figura ilustra los niveles de prueba:



Pruebas de unidad

La meta de las pruebas de unidad es estructural mientras que el otro tipo de pruebas es funcional típico. Como se muestra en el gráfico, en general las funciones son las partes más pequeñas a las que se aplican las pruebas de unidades. En el caso de la orientación a objetos se refiere a los métodos de una clase. La siguiente unidad en tamaño es el módulo, que en nuestro caso se refiere a una clase. Algunas veces, las "combinaciones de módulos" se consideran unidades para los propósitos de las pruebas (estos serían paquetes o subsistemas).

En la base de la pirámide de automatización esta la prueba de unidad (unit test). Las pruebas de unidad deben ser la base de una estrategia de automatización sólida y, como tal, representar la parte más grande de la pirámide.

Las pruebas unitarias son muy efectivas porque aportan datos específicos a un programador (hay un error y está en línea 47), lo que es mucho mejor que tener un mensaje que dice: "Hay un error en cómo se está recuperando registros de miembros de la base de datos ", lo que podría representar 1.000 o más líneas de código. Además, debido a que las pruebas unitarias se escriben normalmente en el mismo lenguaje que la aplicación, los programadores suelen estar más cómodo al escribirlos.

Las unidades a las que se aplican las "pruebas de unidad" son los bloques de construcción de la aplicación, algo así como los ladrillos individuales sobre los que se apoya una casa. Si algunos de estos "ladrillos" son defectuosos, una vez integradas las partes defectuosas en una aplicación, puede tomar mucho tiempo identificarlas y repararlas. Por ello, los bloques de construcción de software deben ser completamente confiables y esta es la meta de las pruebas de unidad.

En términos del PUD (Proceso de Desarrollo Unificado), las pruebas unitarias se llevan a cabo durante las iteraciones de la fase de Elaboración y también en las primeras iteraciones de la Construcción.

Tipos de pruebas de unidad

Generalmente una prueba de unidad consiste en una evaluación estructural (o prueba de caja blanca), lo cual significa que se usan los conocimientos de cómo la unidad es diseñada internamente, y u na evaluación de especificación (o prueba de caja negra), la cual usa lo opuesto: se aplican los test solamente sobre la especificación de los comportamientos de la unidad visibles externamente.

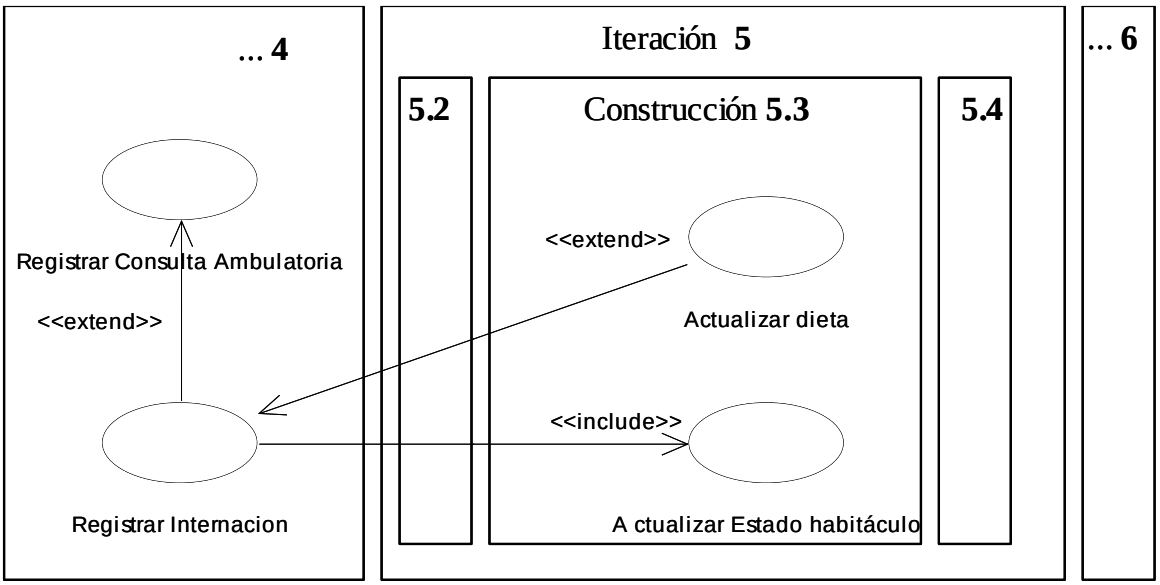
Pruebas de integración

Debido a que las aplicaciones son complejas, deben construirse con partes que primero se desarrollan por separado. La "integración" se refiere a este proceso de ensamble. Se realizan varios tipos de pruebas en los ensambles parciales de la aplicación y en toda ella. La etapa de integración suele producir sorpresas desagradables por incompatibilidad de las partes que se integran. Por esto, el Proceso de Desarrollo Unificado, en particular, intenta evitar la integración "explosiva" mediante la integración continua con múltiples iteraciones.

Cuando se desarrolla la arquitectura, una consideración importante es la facilidad con que se pueden integrar las partes. Sin embargo, rara vez es factible completar los módulos individuales de software antes de la integración. Aunque el proceso de construcción típico tiene la desventaja de trabajar con unidades incompletas, posee la ventaja de ejercer la integración antes en el proceso de desarrollo. Esto ayuda a eliminar los riesgos, al evitar la integración "explosiva".

Las dificultades de integrar las aplicaciones resaltan la importancia de diseñar unidades (como clases y paquetes) que se centren en un propósito lo más que sea posible, disminuyendo sus interfaces mutuas todo lo que se pueda. Estas metas, "alta cohesión" y "bajo acoplamiento", respectivamente, se analizaron en la actividad de diseño. Cuando se aplica a la integración, la verificación se reduce a confirmar que se están uniendo justo los componentes que se planeó ensamblar, justo en la forma que se planeó ensamblarlos. Esa verificación se puede realizar mediante la inspección de los productos de la integración. Realizar pruebas se simplifica con la incorporación de implementaciones de casos de uso completos en cada construcción en lugar de sólo partes de un caso de uso. Crear casos de uso relativamente pequeños desde el principio facilita su ajuste en las construcciones.

Los casos de uso son una fuente ideal de casos de prueba para las pruebas de integración. La idea es que los casos de uso se construyan sobre los que ya están integrados para formar pruebas cada vez más representativas del uso de la aplicación, como se muestra en la siguiente figura:



Relación entre casos de uso, iteraciones y construcciones

Se requiere un número grande de pruebas para validar una aplicación de manera adecuada, y necesitan una organización metódica. Un estilo de organizar los casos de prueba es ponerlos en un paquete en clases especialmente creadas para las pruebas. Una clase o quizá un paquete entero, se puede dedicar a probar toda la aplicación.

Por lo general, las construcciones consisten en el código de varios desarrolladores y es común encontrar muchos problemas cuando se integra el código para crear la construcción. Por ello, se intenta comenzar la integración y las pruebas de integración pronto en el proceso, para ejecutar el código en su contexto final.

Prueba de sistema

La prueba del sistema es la culminación de las pruebas de integración. Consiste en pruebas de caja negra que validan la aplicación completa contra sus requerimientos (funcionales y no funcionales).

Siempre que sea posible, las pruebas del sistema se realizan mientras la aplicación se ejecuta en su entorno requerido. Sin embargo, en ocasiones habrá que conformarse con pruebas de ejecuciones del sistema en un entorno o configuración que no es equivalente a la del cliente.

Bibliografía

- General System Theory; Foundations, Development, Applications. 1968, Ludwig Von Bertalanlfy. Publicado por George Braziller, Nueva York.

- www.itmplatform.com/es/blog/metodologias-agiles-y-clasicas-en-la-gestion-de-un-proyecto/