

Tema 01: Aspectos sobre el diseño de los sistemas.

INTRODUCCION



El diseño de software requiere una serie de técnicas y estándares para su elaboración adecuada. Hoy en día, se emplean técnicas que ayudan a la notación y claridad del diseño, con el fin de que sean transparentes para el usuario como para el ingeniero de software. En esta ocasión se presentarán herramientas para llevar a cabo un mejor diseño.

La notación es muy importante para la comunicación, una notación estandarizada permitirá una comunicación fácil y adecuada entre las partes involucradas en el desarrollo de software. El estándar mundial que se emplea es el UML.

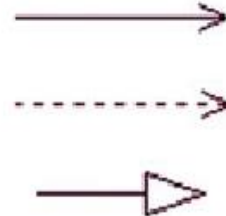
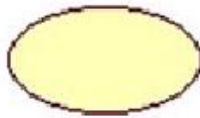


El Lenguaje de Modelamiento Unificado (UML - Unified Modeling Language) es un lenguaje gráfico para visualizar, especificar y documentar cada una de las partes que comprende el desarrollo de software. UML entrega una forma de modelar cosas conceptuales como lo son procesos de negocio y funciones de sistema, además de cosas concretas como escribir clases en

Diagramas de casos de uso

El diagrama de casos de uso representa la forma en cómo un cliente (actor) opera con el sistema en desarrollo, además de la forma, tipo y orden en como los elementos interactúan (operaciones o casos de uso).

ELEMENTOS DE UN DIAGRAMA DE CASO DE USO



Actor

Caso de Uso

Relaciones

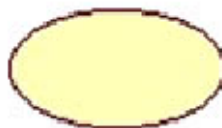
Actor



Una definición previa, es que un actor es un rol que un usuario juega con respecto al sistema. Es importante destacar el uso de la palabra rol, pues con esto se especifica que un actor no necesariamente representa a una persona en particular, sino más bien la labor que realiza frente al sistema.

Como ejemplo a la definición anterior, tenemos el caso de un sistema de ventas en que el rol de vendedor con respecto al sistema puede ser realizado por un vendedor o bien por el jefe de local.

Caso de Uso



Es una operación/tarea específica que se realiza tras una orden de algún agente externo, sea desde una petición de un actor o bien desde la invocación desde otro caso de uso.

Relaciones

Asociación

Es el tipo de relación más básica que indica la invocación desde un actor o caso de uso a otra operación (caso de uso). Dicha relación se denota con una flecha simple.

Dependencia o Instanciación

Es una forma muy particular de relación entre clases, en la cual una clase depende de otra, es decir, se instancia (se crea). Dicha relación se denota con una flecha punteada.

Relaciones

Generalización

Este tipo de relación es uno de los más utilizados, cumple una doble función dependiendo de su estereotipo, que puede ser de Uso (<<uses>>) o de herencia (<<extends>>).

Este tipo de relación está orientado exclusivamente para casos de uso (y no para actores).

Extends: se recomienda utilizar cuando un caso de uso es similar a otro (características).

Relaciones

Use: se recomienda utilizar cuando se tiene un conjunto de características que son similares en más de un caso de uso y no se desea mantener copiada la descripción de la característica.

De lo anterior cabe mencionar que tiene el mismo paradigma en diseño y modelamiento de clases, en donde está la duda clásica de **usar** o **heredar**.

Ejemplo:

Como ejemplo está el caso de una máquina recicladora:

Sistema que controla una máquina de reciclamiento de botellas, tarros y jabas. El sistema debe controlar y/o aceptar:

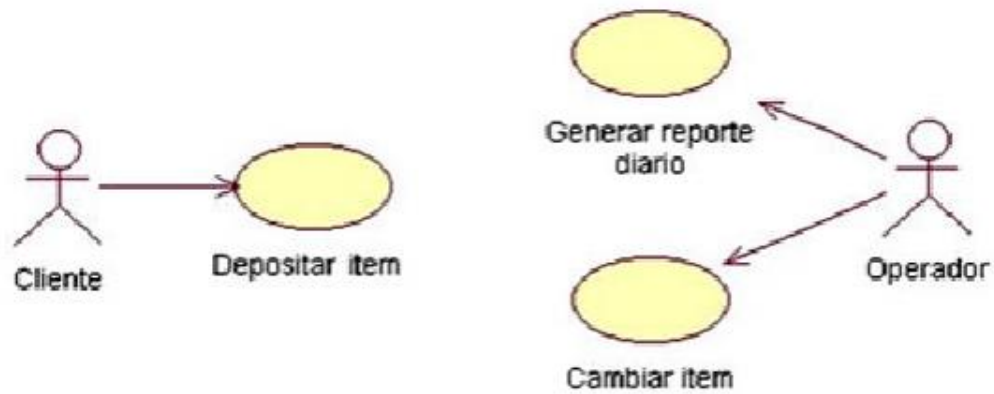
- Registrar el número de ítems ingresados.
- Imprimir un recibo cuando el usuario lo solicita:
 - Describe lo depositado.
 - El valor de cada ítem.
 - Total.
- El usuario/cliente presiona el botón de comienzo.
- Existe un operador que desea saber lo siguiente:
 - Cuántos ítems han sido retornados en el día.
 - Al final de cada día el operador solicita un resumen de todo lo depositado en el día.
- El operador debe además poder cambiar:
 - Información asociada a ítems.
 - Dar una alarma en el caso de que:
 - Ítem se atora.
 - No hay más papel.

DESARROLLO DEL EJEMPLO

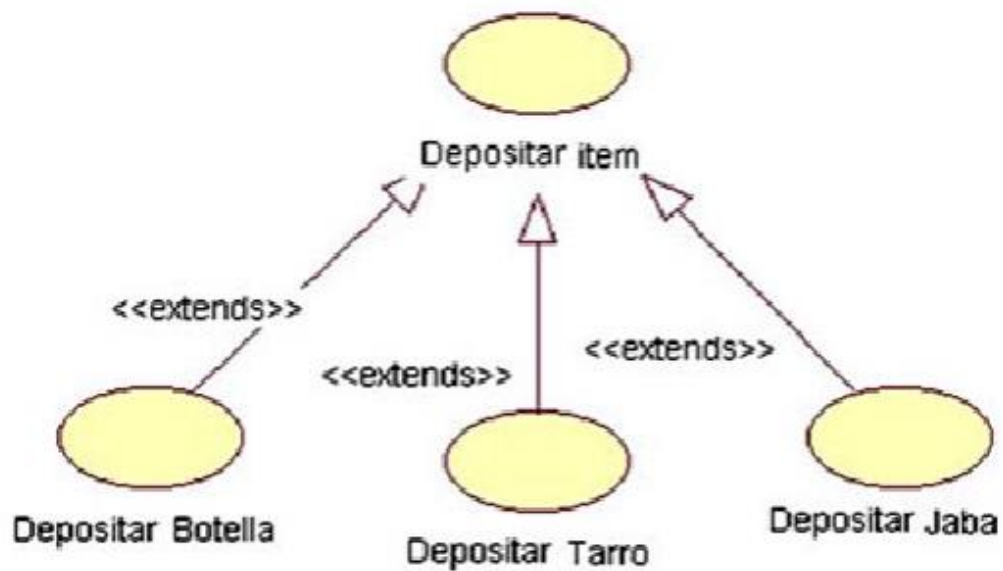
Como una primera aproximación se identifican a los actores que interactúan con el sistema:



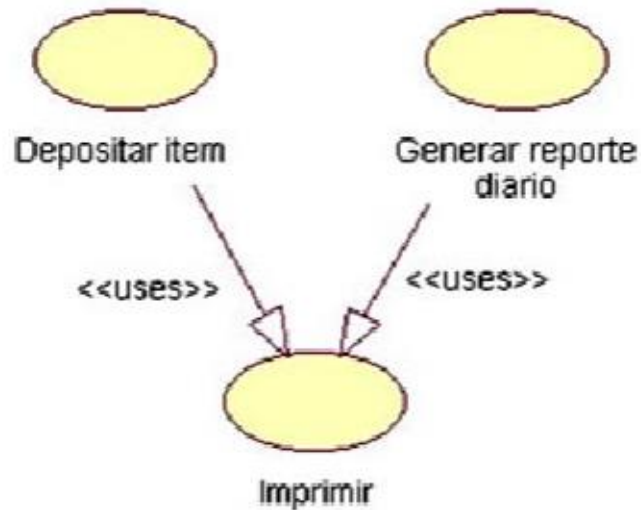
Luego, se sabe que un Cliente puede Depositar Ítems y un Operador puede Cambiar la información de un ítem o bien puede Imprimir un informe:



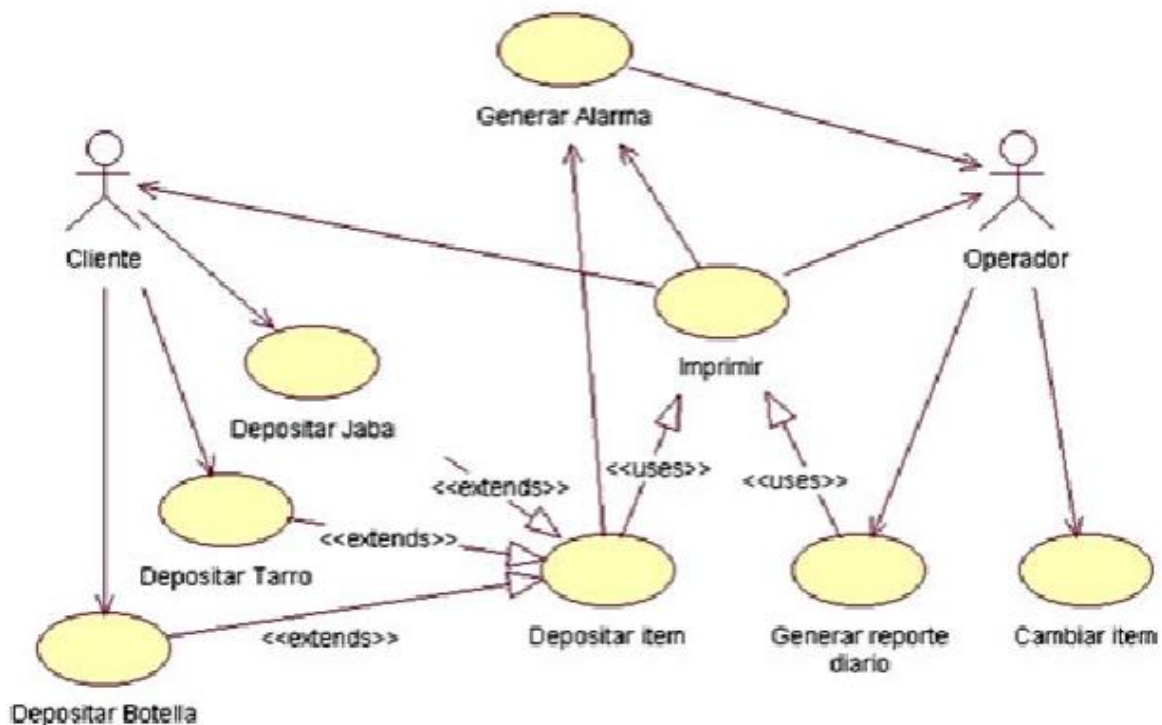
Además se puede notar que un ítem puede ser una botella, un tarro o una jaba:



Otro aspecto es la impresión de comprobantes, que puede ser realizada después de depositar algún ítem por un cliente o bien puede ser realizada a petición de un operador:



Entonces, el diseño completo del diagrama de caso de uso es:

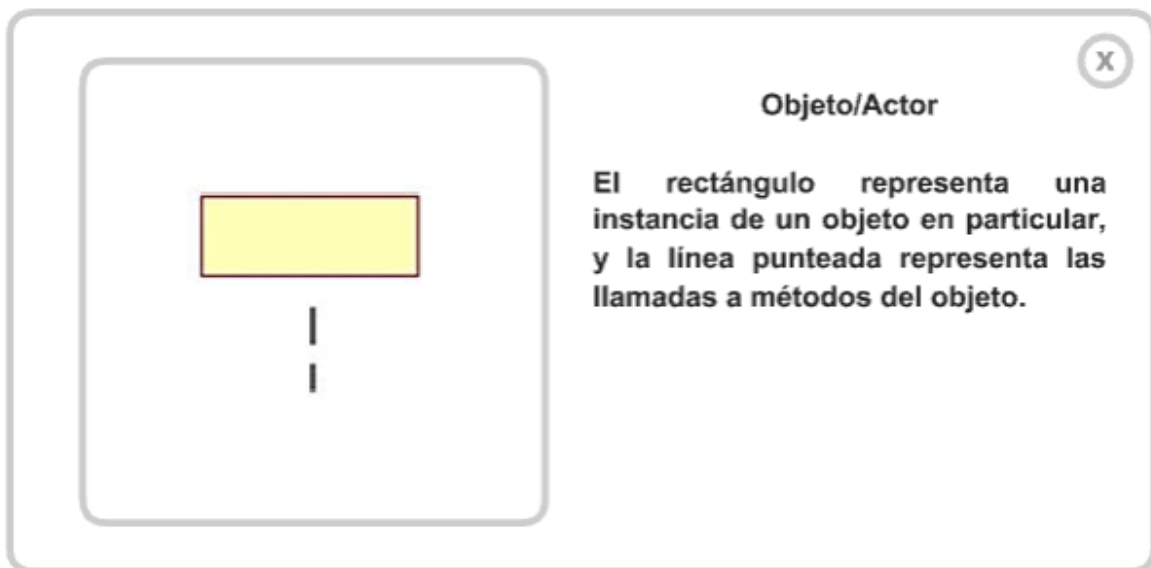


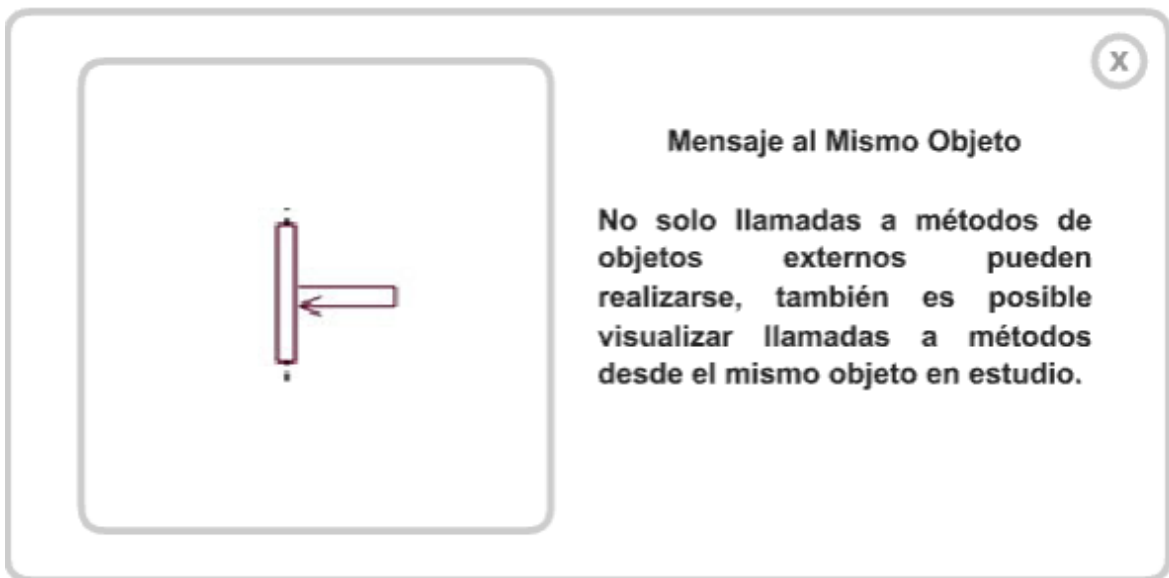
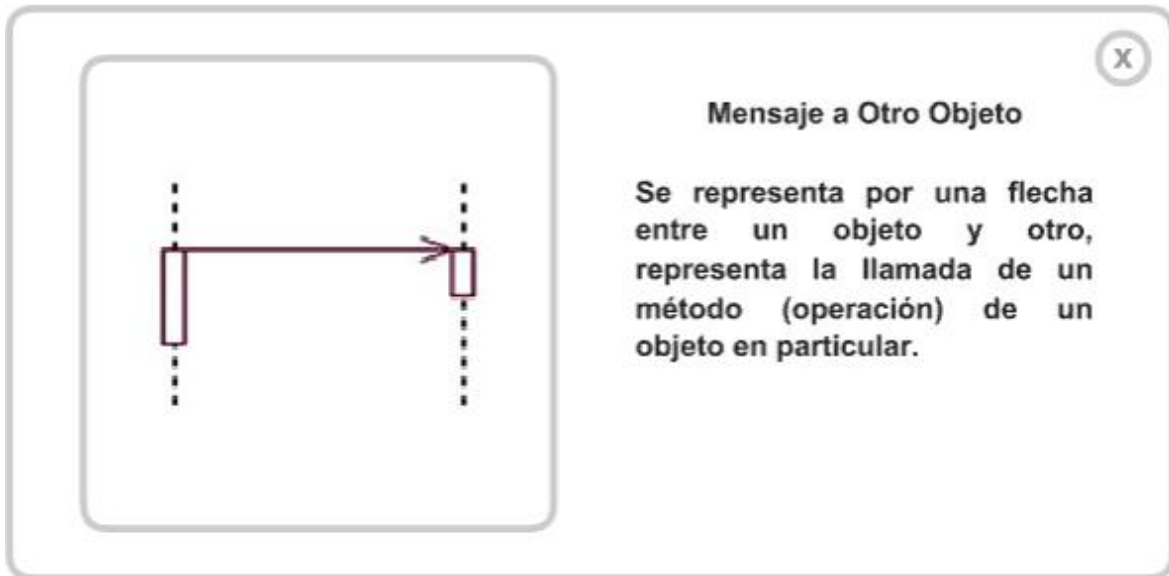
Diagramas de Secuencia

El diagrama de secuencia representa la forma en cómo un cliente (actor) u objetos (clases) se comunican entre sí, como respuesta de una petición a un evento. Esto implica recorrer toda la secuencia de llamadas, de donde se obtienen las responsabilidades claramente.

Dicho diagrama puede ser obtenido de dos partes, desde el Diagrama Estático de Clases o el de Casos de Uso (son diferentes).

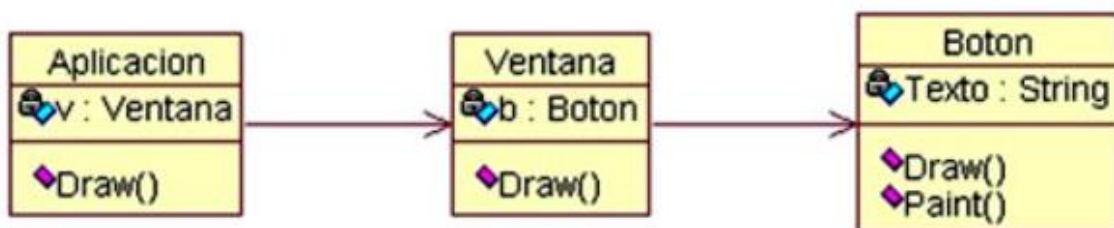
COMPONENTES DE UN DIAGRAMA DE INTERACCIÓN





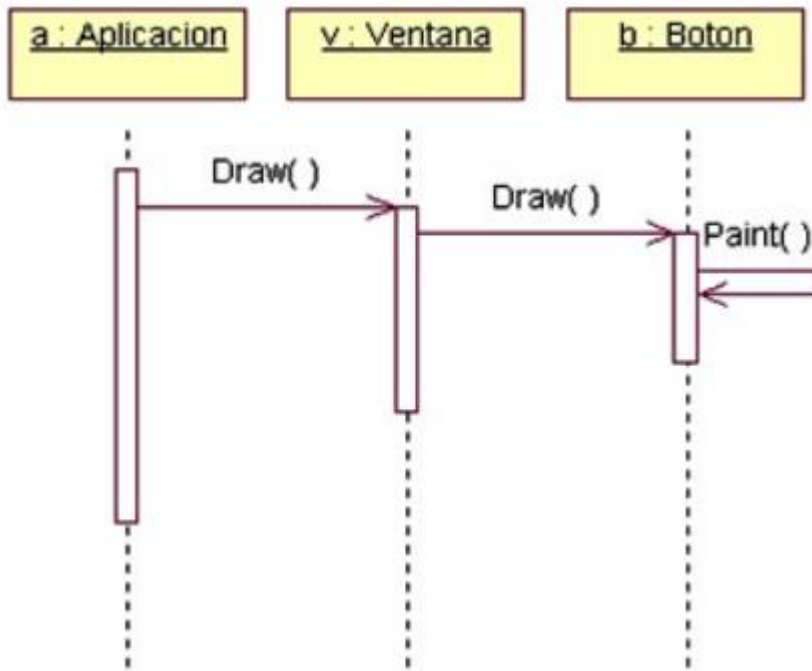
Ejemplo:

En el presente ejemplo, tenemos el diagrama de interacción proveniente del siguiente modelo estático:



Aquí se representa una aplicación que posee una ventana gráfica, y esta a su vez posee internamente un botón.

Entonces el diagrama de interacción para dicho modelo es:



CONTROL DE LECTURA

Lea, analice y responda verdadero o falso según corresponda.

1. En el diagrama de secuencia es posible que se envíe un mensaje de un objeto a sí mismo.

☐ Verdadero

☐ Falso

2. Un actor es un rol que un usuario juega con respecto al sistema.

☐ Verdadero

☐ Falso

3. Se recomienda utilizar extends, cuando un caso de uso es similar a otro.

☐ Verdadero

☐ Falso

4. La generalización es el tipo de relación más básica que indica la invocación desde un actor o caso de uso a otra operación.

☐ Verdadero

☐ Falso

CONTROL DE LECTURA

Lea, analice y responda verdadero o falso según corresponda.

1. En el diagrama de secuencia es posible que se envíe un mensaje de un objeto a sí mismo.

☒ Verdadero

☐ Falso



2. Un actor es un rol que un usuario juega con respecto al sistema.

☒ Verdadero

☐ Falso



3. Se recomienda utilizar extends, cuando un caso de uso es similar a otro.

☒ Verdadero

☐ Falso



4. La generalización es el tipo de relación más básica que indica la invocación desde un actor o caso de uso a otra operación.

☐ Verdadero

☒ Falso



Tema 02: Considerando Objetos. Aspectos básicos para garantizar la adecuada codificación.

INTRODUCCIÓN



El diseño orientado a objetos es una estrategia de diseño en la cual los diseñadores del sistema piensan en términos de “cosas” en lugar de operaciones o funciones. A continuación, se presenta información relacionada al diseño y la implementación de código basada en objetos.

Los sistemas se componen de objetos que interactúan entre ellos y que mantienen su propio estado local y suministran operaciones de esa información. También ocultan información de la representación del estado y, por último, pueden limitar su acceso (características de los objetos).

El proceso de diseño orientado a objetos (DOO) comprende el diseño de clases y las relaciones entre estas clases. Cuando el diseño se implementa como un programa ejecutable, los objetos requeridos se crean dinámicamente utilizando las definiciones de las clases.

DISEÑO ORIENTADO A OBJETOS

El diseño orientado a objetos es parte del desarrollo orientado a objetos en que se utiliza una estrategia en todas las fases del desarrollo:



Análisis orientado a objetos



El diseño orientado a objetos



La programación orientada a objetos

Análisis orientado a objetos

Comprende el desarrollo de un modelo orientado a objetos del dominio de la aplicación. Los objetos identificados reflejan las entidades y operaciones que se asocian con el problema a resolver.



El diseño orientado a objetos



Comprende el desarrollo de un modelo orientado a objetos de un sistema de software para implementar los requerimientos identificados. Los objetos en un DOO están relacionados con la solución del problema por resolver. Pueden existir relaciones cerradas entre algunos objetos del problema y algunos objetos de la solución, pero inevitablemente el diseñador tiene que agregar nuevos objetos para transformar los objetos del problema e implementar la solución..

La programación orientada a objetos

Se refiere a llevar a cabo el diseño de software, utilizando un lenguaje de programación orientado a objetos. Un lenguaje de ese tipo, permite la implementación directa de los objetos y suministra recursos para definir las clases de objetos.



La implementación de UML para manejar adecuadamente el diseño orientado a objetos es el uso del diagrama de clases.

Un diagrama de clases sirve para visualizar las relaciones entre las clases que involucran el sistema, las cuales pueden ser asociativas, de herencia y de uso.

Un diagrama de clases está compuesto por los siguientes elementos

Clase:

- Atributos
- Métodos
- Visibilidad

Relaciones:

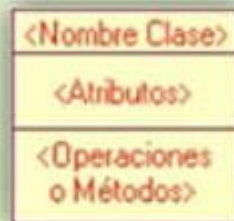
- Herencia
- Composición
- Agregación
- Asociación y uso

Elementos

- Clase

Es la unidad básica que encapsula toda la información de un objeto (un objeto es una instancia de una clase). A través de ella podemos modelar el entorno en estudio (una casa, un auto, una cuenta corriente, etc.)

En UML, una clase es representada por un rectángulo que posee tres divisiones



En donde:

- **Superior:** contiene el nombre de la clase.
- **Intermedio:** contiene los atributos (o variables de instancia) que caracterizan a la clase (pueden ser private, protected o public).
- **Inferior:** contiene los métodos u operaciones, los cuales son la forma como interactúa el objeto con su entorno (dependiendo de la visibilidad: private, protected o public).

Ejemplo:

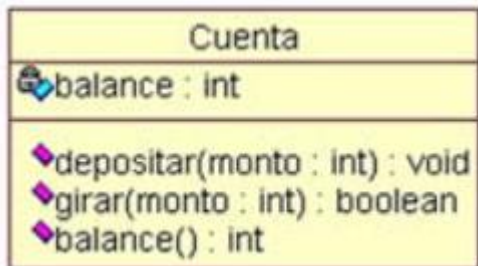
Una cuenta corriente que posee como característica:

- Balance

Puede realizar las operaciones de:

- Depositar
- Girar
- Balance

El diseño asociado es:



ATRIBUTOS Y MÉTODOS


Atributos

Métodos

Atributos




Los atributos o características de una clase pueden ser de tres tipos, los que definen el grado de comunicación y visibilidad de ellos con el entorno. Estos son:

- public (+, 

Métodos



Los métodos u operaciones de una clase son la forma en cómo esta interactúa con su entorno. Estos pueden tener las características:

- public (+, 
 - **Relaciones entre Clases**

Ahora ya definido el concepto de clase, es necesario explicar cómo se pueden interrelacionar dos o más clases (cada uno con características y objetivos diferentes).

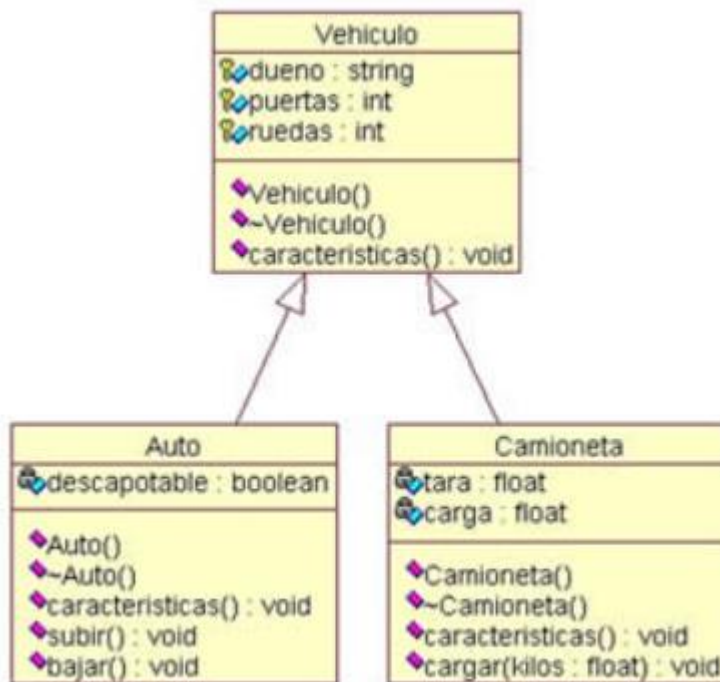
Antes es necesario explicar el concepto de cardinalidad de relaciones: En UML, la cardinalidad de las relaciones indica el grado y nivel de dependencia, se anotan en cada extremo de la relación y estas pueden ser:

- uno o muchos: 1..* (1..n)
- 0 o muchos: 0..* (0..n)
- número fijo: m (m denota el número).

Herencia (Especialización/Generalización):



Indica que una subclase hereda los métodos y atributos especificados por una Super Clase, por ende, la Subclase además de poseer sus propios métodos y atributos, poseerá las características y atributos visibles de la Super Clase (public y protected), ejemplo:



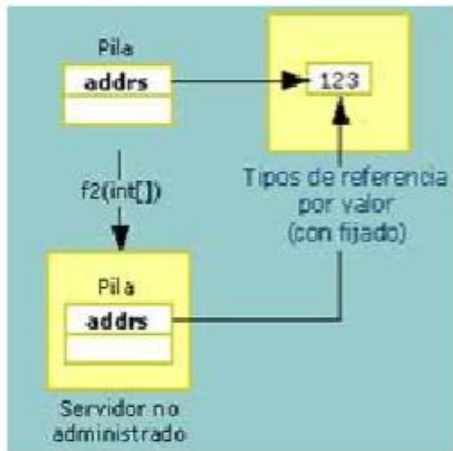
En la figura se especifica que Auto y Camión heredan de Vehículo, es decir, Auto posee las Características de Vehículo (Precio, VelMax, etc.) además posee algo particular que es Descapotable, en cambio Camión también hereda las características de Vehículo (Precio, VelMax, etc.) pero posee como particularidad propia Acoplado, Tara y Carga.

Cabe destacar que fuera de este entorno, lo único "visible" es el método Características aplicable a instancias de Vehículo, Auto y Camión, pues tiene definición pública, en cambio atributos como Descapotable no son visibles por ser privados.

Agregación: 

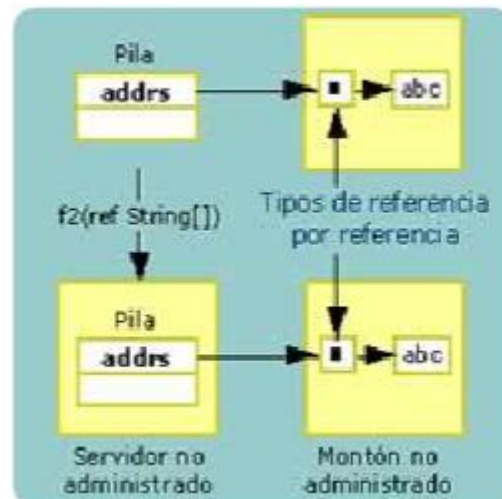
Para modelar objetos complejos, no bastan los tipos de datos básicos que proveen los lenguajes: enteros, reales y secuencias de caracteres. Cuando se requiere componer OBJETOS QUE SON INSTANCIAS DE CLASES definidas por el desarrollador de la aplicación, tenemos dos posibilidades:

OBJETOS QUE SON INSTANCIAS DE CLASES

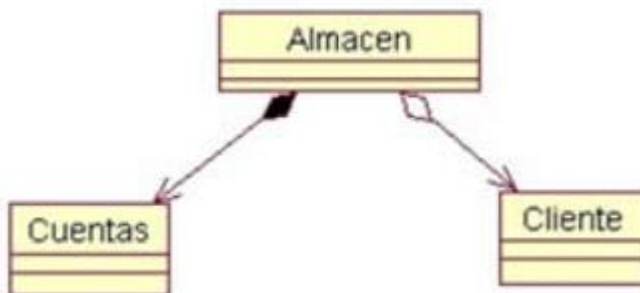


Por Valor: es un tipo de relación estática, en donde el tiempo de vida del objeto incluido está condicionado por el tiempo de vida del que lo incluye. Este tipo de relación es comúnmente llamada Composición (el objeto base se construye a partir del objeto incluido, es decir, es "parte/todo").

Por Referencia: es un tipo de relación dinámica, 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).



Un ejemplo es el siguiente:



En donde se destaca que:

Un Almacén posee Clientes y Cuentas (los rombos van en el objeto que posee las referencias).

Cuando se destruye el objeto Almacén también son destruidos los objetos Cuenta asociados, en cambio no son afectados los objetos Cliente asociados.

La composición (por Valor) se destaca por un rombo relleno.

La agregación (por Referencia) se destaca por un rombo transparente.

La flecha en este tipo de relación indica la navegabilidad del objeto referenciado. Cuando no existe este tipo de particularidad la flecha se elimina.

Asociación: 

La relación entre clases conocida como Asociación, permite asociar objetos que colaboran entre sí. Cabe destacar que no es una relación fuerte, es decir, el tiempo de vida de un objeto no depende del otro.

Ejemplo:



Un cliente puede tener asociadas muchas Órdenes de Compra, en cambio una Orden de Compra solo puede tener asociado un cliente.

Dependencia o Instanciación (uso): 

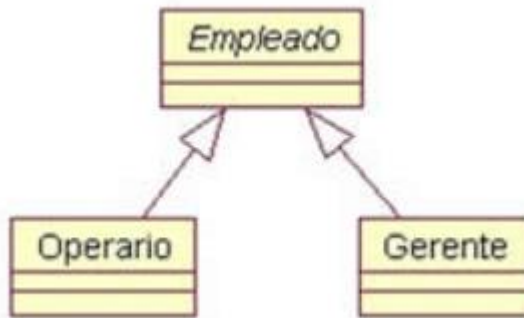
Representa un tipo de relación muy particular, en la que una clase es instanciada (su instanciación es dependiente de otro objeto/clase). Se denota por una flecha punteada.

El uso más particular de este tipo de relación es para denotar la dependencia que tiene una clase de otra, como por ejemplo, una aplicación gráfica que instancia una ventana (la creación del objeto Ventana está condicionado a la instanciación proveniente desde el objeto Aplicación):



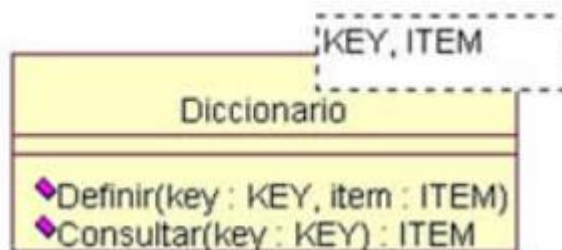
Cabe destacar que el objeto creado (en este caso la Ventana gráfica) no se almacena dentro del objeto que lo crea (en este caso la Aplicación).

- Casos Particulares
 - Clase Abstracta



Una clase abstracta se denota con el nombre de la clase y de los métodos con letra "itálica". Esto indica que la clase definida no puede ser instanciada pues posee métodos abstractos (aun no han sido definidos, es decir, sin implementación). La única forma de utilizarla es definiendo subclases, que implementan los métodos abstractos

- Clase parametrizada



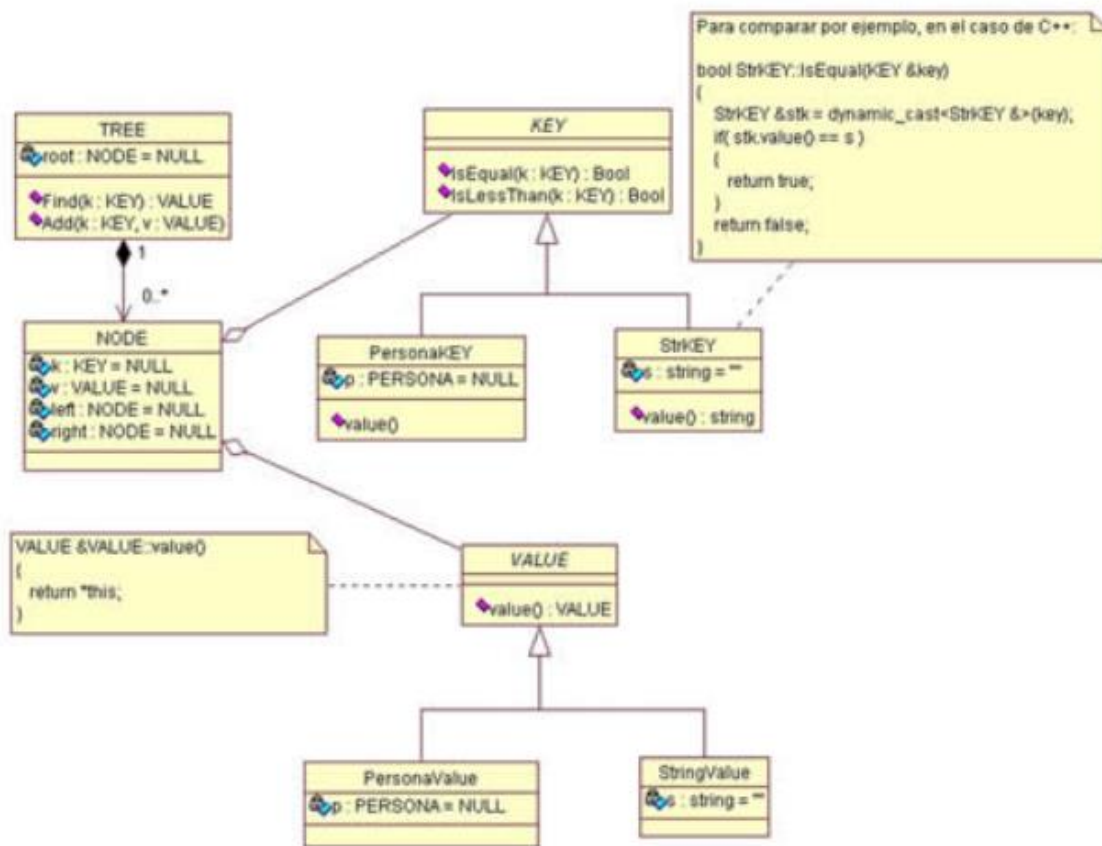
Una clase parametrizada se denota con un subcuadro en el extremo superior de la clase, en donde se especifican los parámetros que deben ser pasados a la clase para que esta pueda ser instanciada. El ejemplo más típico es el caso de un Diccionario en donde una llave o palabra tiene asociado un significado, pero en este caso las llaves y elementos pueden ser

Ejemplo:

Suponga que se tiene el caso de un Diccionario implementado mediante un árbol binario, en donde cada nodo posee:

- key: variable por la cual se realiza la búsqueda, puede ser genérica.
- item: contenido a almacenar en el diccionario asociado a "key", cuyo tipo también puede ser genérico.

Para este caso particular se ha definido un Diccionario para almacenar String y Personas, las cuales pueden funcionar como llaves o como ítems, solo se mostrarán las relaciones para la implementación del Diccionario:



CONTROL DE LECTURA

Lea, analice y responda verdadero o falso según corresponda.

1. La composición (por Valor) se destaca por un rombo transparente.

☐ Verdadero
☐ Falso
2. Los atributos de una clase se orientan a la forma cómo interactúa el objeto con su entorno.

☐ Verdadero
☐ Falso

3. En un diagrama de clases, algunos elementos relacionados con la Clase son: atributos, métodos y visibilidad.

☐ Verdadero

☐ Falso

4. En los métodos, private indica que el método solo será accesible desde dentro de la clase (solo otros métodos de la clase lo pueden acceder).

☐ Verdadero

☐ Falso

CONTROL DE LECTURA

Lea, analice y responda verdadero o falso según corresponda.

1. La composición (por Valor) se destaca por un rombo transparente.

☐ Verdadero

☒ Falso



2. Los atributos de una clase se orientan a la forma cómo interactúa el objeto con su entorno.

☐ Verdadero

☒ Falso



3. En un diagrama de clases, algunos elementos relacionados con la Clase son: atributos, métodos y visibilidad.

☒ Verdadero

☐ Falso



4. En los métodos, private indica que el método solo será accesible desde dentro de la clase (solo otros métodos de la clase lo pueden acceder).

☒ Verdadero

☐ Falso



Tema 03: Escribiendo código fuente. La prueba de los Programas.

INTRODUCCIÓN



El diseño orientado a objetos es una estrategia de diseño en la cual los diseñadores del sistema piensan en términos de “cosas” en lugar de operaciones o funciones. A continuación, se presenta información relacionada al diseño y la implementación de código basada en objetos.

La codificación de software es realmente un arte que permite la creación a partir de técnicas, estándares, métricas y conocimientos, pero está muy de la mano con la creatividad de los ingenieros de software para poder crear basado en los requerimientos.

Así como es un arte, también es un proceso sistemático y disciplinado en el cual se incluyen muchos estándares para cada uno de los aspectos, a continuación se presentan algunos elementos que permitirán la creación de software con alta calidad.

Código fuente

Muchos desarrolladores de software hacen código en una forma desordenada, colocando las cosas de acuerdo a como vayan apareciendo o como se las vayan imaginando, de acuerdo al principio:

Si funciona (corre) esta bueno

Realmente es válido pensar de esa forma, a corto plazo no habrá problema, sin embargo en las fases de mantenimiento (tema a tratarse en la tercera unidad) será un verdadero dolor de cabeza o un gasto de recursos, puesto que es será mucho mas fácil encontrar una falla o realizar cambios sobre un código ordenado que sobre una mar de líneas de código.

Una recomendación para el ordenamiento de código en orden superior a inferior es la siguiente:

- Variables locales o públicas
- Funciones o procedimientos
 - Primero los públicos
 - Segundo los privados
- Códigos de eventos (botones, cajas de texto, cuadrículas, etc.)
- Códigos que se ejecutan muy raramente

Los códigos deberán estar (siempre que el lenguaje de programación lo permita) con manejo de excepciones o control de errores, un segmento de código que no esté encerrado se considerará un problema.

Es recomendable establecer estándares en los nombres de los identificadores de objeto, por ejemplo el uso de prefijos, en donde para llamar una variable se antepone el prefijo **var_** así por ejemplo, dada una variable edad, el resultado sería: **var_edad** , de igual forma para elementos como las clases se podría usar un prefijo **c** , por ejemplo, dada la clase Empleado, el nombre sería **cEmpleado**.

El establecimiento de comentarios es muy importante, permitiendo

Establecer la fecha y autor de cada objeto creado (lo que garantiza un grado de responsabilidad sobre los objetos y desarrolladores)

Comentar segmentos de código complejo, para que en el futuro su análisis no dependa de "Descifrar" lo que se hizo

Descifrar" lo que se hizo Establecer referencias o aclaraciones sobre llamadas a métodos, otros sistemas, otras bases de datos, ficheros, etc.

Interfaces

Es altamente recomendable que se establezca un estándar en el diseño de las interfaces, que permita que al desarrollar software en equipo, no se tenga una mezcla de diseños.

LOS ESTÁNDARES QUE SE RECOMIENDAN

- Colores de los formularios o páginas web (de acuerdo al tipo de software que se esté desarrollando)
- Nombres de los controles a emplear. (cajas de texto, etiquetas, botones, listas desplegables, etc.)
- Posición de los objetos, muchos desarrolladores en una pantalla colocan los botones de acción a la izquierda y en otras a la derecha. Esto tiende a confundir a los usuarios.

- **Uso adecuado de mensajes para la visualización de pasos, advertencias o indicaciones.**
- **Establecimiento adecuado de colores. Muchos desarrolladores usan colores intensos, olvidando que los usuarios probablemente estarán frente al monitor y frente al software por horas, este tipo de interfaz cansa la vista, generando malhumor en los usuarios y resistencias al uso de la aplicación.**

Bases de datos

A parte de la aplicación de las formas normales en el diseño de la base de datos, se recomiendan algunos elementos básicos pero que resultan de mucha utilidad



☐ **Usar nombres fáciles de recordar para las tablas**

☐ **Mucha atención al tamaño máximo de los campos para almacenar cadenas de texto**

☐ **Cuidado en la integridad referencial**

Pruebas en el desarrollo

La primera línea de prueba para las aplicaciones son los desarrolladores, por lo que cuando se programa o se está supervisando a los desarrolladores, deberán probarse cosas como:

Verificar caracteres máximos en las cajas de texto.

Probar entradas equivocadas, por ejemplo negativos, valores con cero (0), valores vacíos, valores extremadamente largos.

Trate de dar clic muchas veces en un botón.

Verifique que sucede si cierra la aplicación a la mitad de un proceso.

Consulte grandes cantidades de datos.

Conecte su aplicación en más de un equipo y pruebe cómo se comporta.

Pruebe que sucede si cambia la resolución de la pantalla.

En la tercera unidad, se presentará el proceso formal de pruebas de software.

CONTROL DE LECTURA

Lea, analice y responda verdadero o falso según corresponda.

1. Usar nombres fáciles de recordar para las tablas es una recomendación que se presenta para el diseño de interfaces

☐ Verdadero

☒ Falso



2. Se recomienda que los Códigos de eventos (botones, cajas de texto, cuadrículas, etc.) sean lo primero que se encuentre en el código de los formularios o páginas web

☐ Verdadero

☒ Falso



3. Una prueba en el momento del desarrollo es probar entradas equivocadas, por ejemplo negativos, valores con cero (0), valores vacíos, valores extremadamente largos

☒ Verdadero

☐ Falso



4. Se recomienda poner atención en la creación de interfaces y en el uso adecuado de mensajes para la visualización de pasos, advertencias o indicaciones.

☒ Verdadero

☐ Falso



CONTROL DE LECTURA

Lea, analice y responda verdadero o falso según corresponda.

1. Usar nombres fáciles de recordar para las tablas es una recomendación que se presenta para el diseño de interfaces

☐ Verdadero

☒ Falso



2. Se recomienda que los Códigos de eventos (botones, cajas de texto, cuadrículas, etc.) sean lo primero que se encuentre en el código de los formularios o páginas web

☐ Verdadero

☒ Falso



3. Una prueba en el momento del desarrollo es probar entradas equivocadas, por ejemplo negativos, valores con cero (0), valores vacíos, valores extremadamente largos

☒ Verdadero

☐ Falso



4. Se recomienda poner atención en la creación de interfaces y en el uso adecuado de mensajes para la visualización de pasos, advertencias o indicaciones.

☒ Verdadero

☐ Falso



Tema 04: Discusión general del ciclo de vida del software hasta esta etapa.

INTRODUCCIÓN



El ciclo de vida de software ha sido analizado anteriormente, sus detalles se han ido presentando en diferentes clases. En esta ocasión es buen momento para hacer una revisión a las fases que hasta la fecha han sido estudiadas, así como establecer los elementos que continúan en el ciclo de lo que es el desarrollo de software.

Hasta esta fase, hemos analizado los elementos de los requerimientos, el diseño, aplicando técnicas de UML y hemos visto algunos elementos de la codificación, aplicando estándares.

Hemos ido en las fases del ciclo de vida de software, pero, ¿qué es el ciclo de vida?

**“Una aproximación lógica a la adquisición, el suministro,
el desarrollo, la explotación y el mantenimiento del software”.
IEEE 1074**

**“Un marco de referencia que contiene los procesos, las actividades
y las tareas involucradas en el desarrollo, la explotación y el
mantenimiento de un producto de software, abarcando la vida del
sistema desde la definición de los requisitos hasta la finalización de
su uso”.
ISO 12207-1**

Si analizamos el ciclo de vida, llevado hasta este momento, diremos que en un inicio hemos visto lo importante de recabar bien los requerimientos, desarrollando entrevistas, revisión de documentos, análisis de procesos, revisión de sistemas actuales, entre otras cosas.

Es conveniente que cuando se hace el acuerdo de los requerimientos se haga un acuerdo (contrato) en el cual ambas partes (desarrollador y cliente) certifiquen que están de acuerdo.

DESARROLLO DE SOFTWARE EXTERNO O INTERNO

**La visión para este contrato,
puede aplicarse al desarrollo de
software externo o interno.**

Desarrollo interno

Desarrollo externo

Desarrollo interno

Orientado a empleados del área de ingenieros desarrolladores de software de una empresa/institución que hacen software para otras áreas de la misma empresa.

Desarrollo externo

Relacionado a software que se crea por parte de una empresa externa y que servirá para una o más áreas de la empresa.

[Ver Modelo de Contrato.](#)

Posteriormente, se ha considerado el aspecto de diseño, el cual debe ser siempre considerado como parte esencial, puesto que en el diseño se hará el planteamiento adecuado de lo que se espera desarrollar.

La utilización de herramientas de notación estandarizadas mundialmente, (UML, por ejemplo), proporcionan a los desarrolladores un enfoque profesional y que dará como resultado un desarrollo con mejores garantías de comprensión entre ambas partes.

Luego, se ha trabajado en la parte de

Tema 05: Paradigmas de desarrollo y la planificación de proyectos.

INTRODUCCIÓN

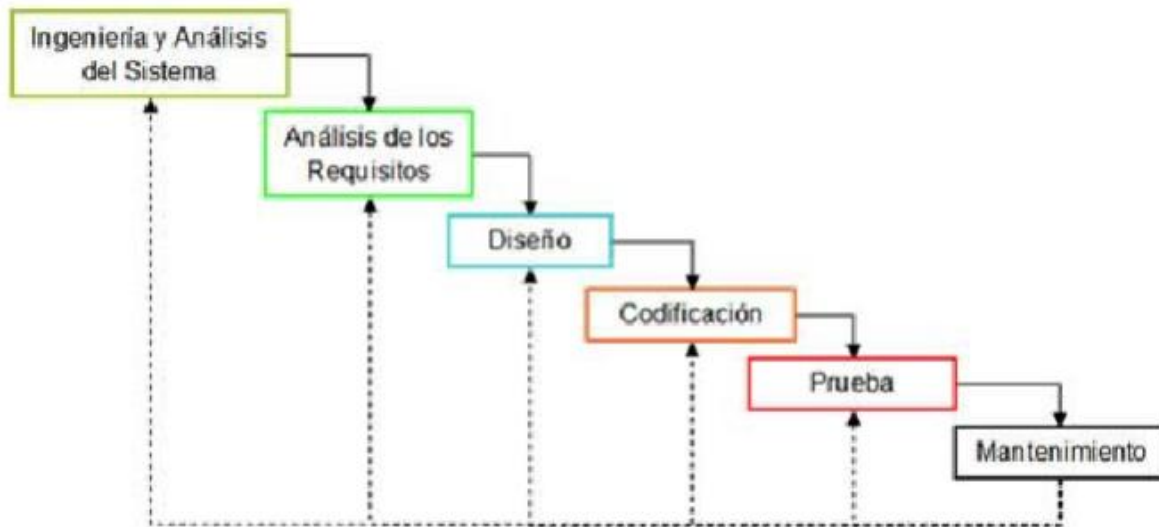


El diseño de software requiere una serie de técnicas y estándares para su elaboración adecuada. Hoy en día, se emplean técnicas que ayudan a la notación y claridad del diseño, con el fin de que sean transparentes para el usuario como para el ingeniero de software. En esta ocasión se presentarán herramientas para llevar a cabo un mejor diseño.

MODELO EN CASCADA

El modelo clásico del proceso de desarrollo de software es el modelo en cascada, también llamado modelo lineal secuencial. Este es una secuencia de actividades que consiste en el análisis de requerimientos, el diseño, la implementación, la integración y las pruebas. Estas etapas en realidad no se ejecutan en una secuencia estricta, ya que suele ser poco práctico completar totalmente una de estas etapas antes de comenzar la otra.

MODELO EN CASCADA



Ingeniería y análisis del sistema

Debido a que el software es siempre parte de un sistema mayor, el trabajo comienza estableciendo los requisitos de todos los elementos del sistema y luego asignando algún subconjunto de estos requisitos al software.

Análisis de los requisitos del software

El proceso de recopilación de los requisitos se centra e intensifica especialmente en el software. El ingeniero de software (analistas) debe comprender el ámbito de la información del software, así como la función, el rendimiento y las interfaces requeridas.

Diseño

El diseño del software se enfoca en cuatro atributos distintos del programa: la estructura de los datos, la arquitectura del software, el detalle procedimental y la caracterización de la interfaz. El proceso de diseño traduce los requisitos en una representación del software con la calidad requerida antes de que comience la codificación.

Codificación

El diseño debe traducirse en una forma legible para la máquina. El paso de codificación realiza esta tarea. Si el diseño se realiza de una manera detallada la codificación puede realizarse mecánicamente.

Prueba

Una vez que se ha generado el código comienza la prueba del programa. La prueba se centra en la lógica interna del software, y en las funciones externas, realizando pruebas que aseguren que la entrada definida produce los resultados que realmente se requieren.

Mantenimiento

El software sufrirá cambios después de que se entrega al cliente. Los cambios ocurrirán debido a encontrar errores, a que el software deba adaptarse a cambios del entorno externo (sistema operativo o dispositivos periféricos), o debido a que el cliente requiera ampliaciones funcionales o del rendimiento.

Desventajas:

- Los proyectos reales raramente siguen el flujo secuencial que propone el modelo, siempre hay iteraciones y se crean problemas en la aplicación del paradigma.
- Normalmente es difícil para el cliente establecer explícitamente al principio todos los requisitos. El ciclo de vida clásico lo requiere y tiene dificultades en acomodar posibles incertidumbres que pueden existir al comienzo de muchos productos.
- El cliente debe tener paciencia. Hasta llegar a las etapas finales del proyecto, no estará disponible una versión operativa del programa. Un error importante no detectado hasta que el programa esté funcionando puede ser desastroso.

MODELO EN ESPIRAL y MODELO BASADO EN PROTOTIPOS

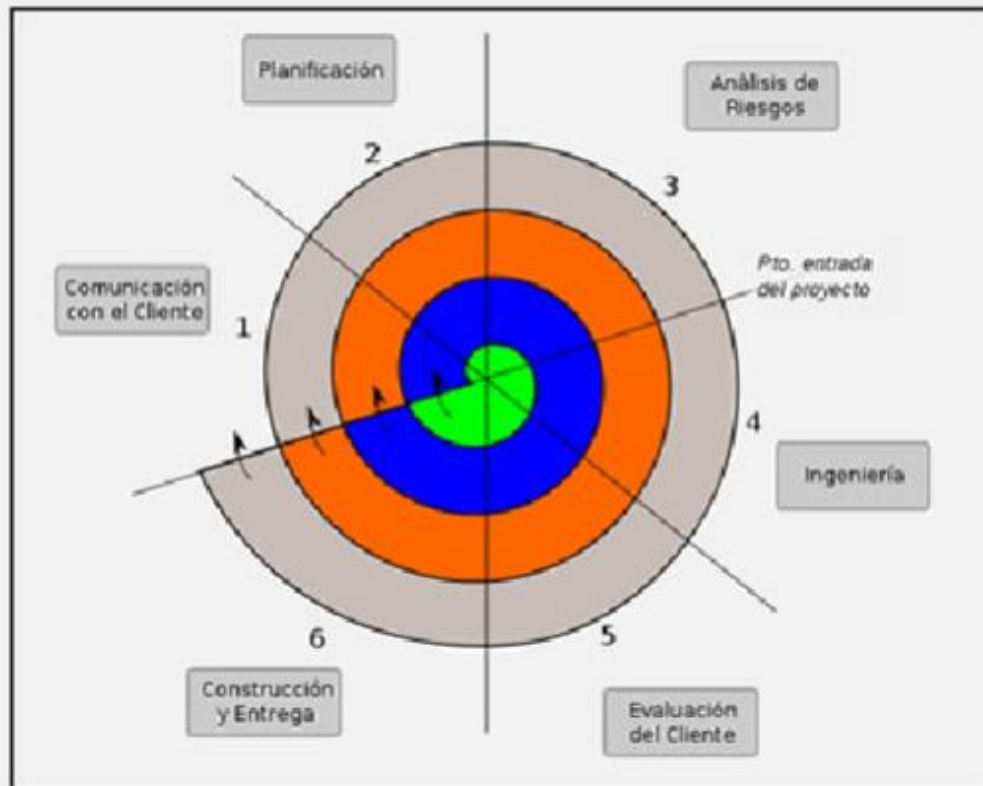
**Modelo
en Espiral**

**Modelo basado en
Prototipos**

**Modelo de
desarrollo orientado
a la reutilización**



Modelo en Espiral



Modelo en Espiral



El modelo espiral es un modelo de proceso de software evolutivo que conjuga la naturaleza iterativa de construcción de prototipos con los aspectos controlados y sistemáticos del modelo lineal secuencial.

Proporciona el potencial para el desarrollo rápido de versiones incrementales del software. En este modelo, el software se desarrolla en una serie de versiones incrementales.

A diferencia del modelo de proceso clásico que termina cuando se entrega el software, el modelo en espiral puede adaptarse y aplicarse a lo largo de la vida del software de computadora.

Este modelo se divide en un número de actividades de marco de trabajo, también llamadas regiones de tareas.

Modelo en Espiral



Generalmente existen tres y seis regiones de tareas:

- Comunicación con el cliente.
- Planificación.
- Análisis de riesgo.
- Ingeniería.
- Construcción y acción.
- Evaluación del cliente.

Cada una de estas regiones está compuesta por un conjunto de tareas del trabajo llamado conjunto de tareas, que se adaptan a las características del proyecto que se va a emprender.

Modelo basado en Prototipos



Modelo basado en Prototipos



Un cliente, a menudo, define un conjunto de objetivos generales para el software, pero no identifica los requisitos detallados de entrada, proceso o salida.

El responsable del desarrollo del software puede no estar seguro de la eficacia de un algoritmo, de la capacidad de adaptación de un sistema operativo, o de la forma en que debería tomarse la interacción hombre-máquina.

En estas y en otras muchas situaciones, un paradigma de construcción de prototipos puede ofrecer el mejor enfoque.

El paradigma de construcción de prototipos comienza con la recolección de requisitos.

Modelo basado en Prototipos



El desarrollador y el cliente encuentran y definen los objetivos globales para el software, e identifican los objetivos conocidos y las áreas del esquema donde es obligatoria más definición.

Entonces aparece un **“diseño rápido”**.

El diseño rápido lleva a la construcción de un prototipo. El prototipo lo evalúa el **cliente/usuario** y se utiliza para refinar los requisitos del software a desarrollar.

Modelo de desarrollo orientado a la reutilización



En este proceso se definen los roles involucrados que participan en la identificación de la reutilización en cada uno de los elementos de software.

Cada rol que participa dentro del proceso de reutilización debe participar en definir los elementos reutilizables en función a las actividades que realiza en el proceso de soluciones de software.

Este modelo se basa en software o componentes de software existentes y es usado muchas veces cuando un nuevo software absorbe diseños antiguos o que han crecido tanto que ha quedado limitado.

CONTROL DE LECTURA

Lea, analice y responda verdadero o falso según corresponda.

1. Una desventaja del modelo en cascada es que los proyectos reales raramente siguen el flujo secuencial que propone el modelo, siempre hay iteraciones y se crean problemas en la aplicación del paradigma.

☒ Verdadero

☐ Falso

2. El modelo en prototipos es un modelo de proceso de software evolutivo que conjuga la naturaleza iterativa de construcción de prototipos con los aspectos controlados y sistemáticos del modelo lineal secuencial.

☐ Verdadero

☒ Falso

3. Con el modelo en prototipos, se puede presentar un diseño rápido para la evaluación del cliente.

☒ Verdadero

☐ Falso

4. La ventaja del modelo de reutilización radica en su sencillez, ya que sigue los pasos intuitivos necesarios a la hora de desarrollar el software.

☐ Verdadero

☒ Falso

CONTROL DE LECTURA

Lea, analice y responda verdadero o falso según corresponda.

1. Una desventaja del modelo en cascada es que los proyectos reales raramente siguen el flujo secuencial que propone el modelo, siempre hay iteraciones y se crean problemas en la aplicación del paradigma.

☒ Verdadero

☐ Falso



2. El modelo en prototipos es un modelo de proceso de software evolutivo que conjuga la naturaleza iterativa de construcción de prototipos con los aspectos controlados y sistemáticos del modelo lineal secuencial.

☐ Verdadero

☒ Falso



3. Con el modelo en prototipos, se puede presentar un diseño rápido para la evaluación del cliente.

☒ Verdadero

☐ Falso



4. La ventaja del modelo de reutilización radica en su sencillez, ya que sigue los pasos intuitivos necesarios a la hora de desarrollar el software.

☐ Verdadero

☒ Falso

