PROCESO DIRECCIÓN DE FORMACIÓN PROFESIONAL INTEGRAL

FORMATO GUÍA DE APRENDIZAJE

IDENTIFICACIÓN DE LA GUIA DE APRENDIZAJE

- Denominación del Programa de Formación: ANALISIS Y DESARROLLO DE SISTEMAS DE INFORMACION.
- Código del Programa de Formación: 228106
- Nombre del Proyecto: SISTEMA INTEGRAL WEB PARA GESTION DE PROCESOS EDUCATIVOS DEL CEET
- Fase del Proyecto: PLANEACIÓN
- Actividad de Proyecto: ESPECIFICAR MODELO CONCEPTUAL DEL SISTEMA, ESPECIFICANDO NECESIDADES Y REQUERIMIENTOS.
- Competencia: ANALIZAR LOS REQUISITOS DEL CLIENTE PARA CONSTRUIR EL SISTEMA DE INFORMACIÓN
- Resultados de Aprendizaje Alcanzar: CONSTRUIR EL MODELO CONCEPTUAL DEL MACROSISTEMA FRENTE A LOS REQUERIMIENTOS DEL CLIENTE, MEDIANTE EL USO E INTERPRETACIÓN DE LA INFORMACIÓN LEVANTADA, REPRESENTADO EN DIAGRAMAS DE CLASE, DE INTERACCIÓN, COLABORACIÓN Y CONTRATOS DE OPERACIÓN, DE ACUERDO CON LAS DIFERENTES SECUENCIAS, FASES Y PROCEDIMIENTOS DEL SISTEMA.
- Duración de la Guía

2. PRESENTACIÓN

"UML es el acrónimo en inglés para “lenguaje unificado de modelado”. Pero, ¿qué significa modelar? La respuesta corta sería: “construir modelos”. Sin embargo, esta respuesta nos lleva a otra pregunta: ¿qué es un modelo? La palabra modelo tiene varias acepciones en español. Para nosotros y en el contexto de UML, un modelo “es una descripción analógica para ayudar a visualizar algo que no se puede observar directamente y que se realiza con un propósito determinado y se destina a un público específico”.

MODELOS DE SOFTWARE

Los modelos de software tienen las siguientes características:

- Descripción analógica: el modelo no es el sistema de software en sí, sino su representación
- De algo que no se puede ser observar directamente: el software nunca puede ser observado directamente porque es intangible e invisible por su propia naturaleza; de hecho, los modelos son la única manera de “observar” el software.
- Se realizó con cierto propósito: un modelo de software puede servir, entre otras cosas, para construir una aplicación todavía inexistente, para validar conceptos con otros interesados en el desarrollo o para documentar un programa existente con el fin de facilitar su mantenimiento.
- Se destina a un determinado público: puede ser un modelo para usuarios finales, analistas, programadores, testers, etc.

Por lo tanto, un mismo sistema puede tener varios modelos, que dependen del propósito y del público al que se dirige.

Pero el modelado de software tiene una complejidad adicional. Según varias definiciones, el software es en sí un modelo de la realidad. Si así fuera, y esta cuestión es un tanto filosófica, un modelo de software es el modelo de un modelo; es decir, un meta-modelo. Puede que estemos o no de acuerdo con esta apreciación, pero lo cierto es que, en muchos casos, esto se convierte en una complejidad adicional.

POR QUÉ EL SOFTWARE NECESITA MODELOS

El software necesita modelos por las mismas razones que cualquier otra construcción humana: para comunicar de manera sencilla una idea abstracta, existente o no, o para describir un producto existente.

En efecto, el modelo más detallado de un producto de software es el código fuente. Pero es como decir que el mejor modelo de un edificio es el edificio mismo; esto no nos sirve para concebirlo antes de la construcción ni para entender sus aspectos más ocultos con vistas al mantenimiento.

Sin embargo, en el software el modelado es aún más importante que en las otras ingenierías. Esto tiene varias razones de ser:

- El software es invisible e intangible: sólo se ve su comportamiento, sus efectos en el medio.
- El software es mucho más modificable que otros productos realizados por el hombre: esta modificabilidad es percibida por los ajenos a la industria, lo que provoca que haya un incentivo mucho más fuerte para pedir modificaciones.
- El software se desarrolla por proyectos, no en forma repetitiva como los productos de la industria manufacturera. Esto hace que cada vez que construyamos un producto de software estemos enfrentándonos a un problema nuevo.
- El software es substancialmente complejo,1 con cientos o miles de partes interactuando, diferentes entre sí, y que pueden ir cambiando de estados a lo largo de su vida: esto hace que analizar un producto de software requiera mecanismos de abstracción y de un lenguaje para representarlo.
- El desarrollo del software es inherentemente complejo: la complejidad del producto lleva a la complejidad de los proyectos, de los equipos de desarrollo y de la administración de proyectos.

UML

UML es una notación de modelado visual, que utiliza diagramas para mostrar distintos aspectos de un sistema. Si bien muchos destacan que UML es apto para modelar cualquier sistema, su mayor difusión y sus principales virtudes se advierten en el campo de los sistemas de software. Esto no obsta para que muchos profesionales intenten usar UML en situaciones diversas, haciendo uso de esa máxima que dice que “cuando la única herramienta que conocemos es el martillo, aun los tornillos nos parecen clavos”.

Surgió en 1995, por iniciativa de Grady Booch, James Rumbaugh e Ivar Jacobson, tres conocidos ingenieros de software que ya habían avanzado con sus propias notaciones de modelado. Precisamente, UML se define como “unificado”, porque surgió como síntesis de los mejores elementos de las notaciones previas.

Y nos ha venido muy bien, ya que, a mediados de la década de 1990, nos encontrábamos empantanados en la falta de un estándar, aunque fuese de facto, que marcase el camino para la modelización de software orientado a objetos.

Luego UML se especificó con más rigurosidad y, en 1997, se presentó la versión 1.0, que fue aprobada y establecida como estándar por el OMG (Object Management Group, un consorcio de empresas de desarrollo de estándares). De allí en más, siguió evolucionando, formalizándose y –lo que no siempre es una ventaja– creciendo y complejizándose.

Hacia 2000, UML ya se había convertido en el estándar de facto para modelización de software orientado a objetos.

En la actualidad, UML es un lenguaje de visualización, especificación y documentación de software, basado en trece tipos de diagramas, cada uno con sus objetivos, destinatarios y contexto de uso.

Se habla de lenguaje, en cuanto a que es una herramienta de comunicación formal, con una serie de construcciones, una sintaxis y una semántica definidas. Así, os elementos constructivos son diagramas y sus partes, la sintaxis es la descripción de cómo deben realizarse esos diagramas y la semántica define el significado de cada diagrama y elemento de los mismos.

La palabra “lenguaje” puede resultar extraña en el contexto de los ingenieros de software, pero debemos acostumbrarnos a ella porque la vamos a usar a lo largo del libro.

Además, UML es extensible. Se han definido varios mecanismos de extensibilidad, que permiten aumentar usos de UML.

De hecho, UML suele ser bastante más amplio de lo que solemos necesitar en ocasiones. Los creadores de la notación llegan a afirmar el 80% de la mayoría de los sistemas se puede modelar con el 20% de las construcciones de UML2. Los diagramas, cuando sea necesario, se pueden agregar algunos dibujos, texto y notas, que los hagan más claros.

Es importante destacar que la naturaleza de modelo basado en diagramas de UML no le impide tener una definición formal. Ya, en 1997, se formó un grupo denominado pUML,3 que reunió a desarrolladores e investigadores para convertir a UML en un lenguaje bien definido y riguroso. Luego, el OMG adoptó el lenguaje gráfico MOF(acrónimo de Meta Object Facility) y el lenguaje textual basado en lógica de primer orden OCL (acrónimo de Object Constraint Language), que se usan para definir varios lenguajes de modelado, entre ellos UML. OCL se usa también en conjunto con UML para expresar restricciones de implementación.

Para qué usar UML

Hay varios usos que se pueden hacer de UML, pero en haras de clasificar, podemos distinguir dos:

- Como herramienta de comunicación entre humanos.
- Como herramienta de desarrollo.

En el primer caso, usamos UML para mejorar el entendimiento de alguno o varios aspectos dentro del equipo de desarrollo, entre el equipo de desarrollo y otros interesados en el proyecto, o para documentar aspectos del desarrollo para el mantenimiento posterior del sistema.

UML y la orientación a objetos

La palabra “unificado” dentro del acrónimo UML ha llevado a muchos a tratar de usar UML para modelar cualquier tipo de software, independientemente del paradigma de desarrollo.

Lo cierto es que todo puede hacerse. Sin embargo, no hay que olvidar que UML surgió en el marco del paradigma orientado a objetos, por lo que se aplica más naturalmente a ellos. Por esta razón, analizaremos a UML solamente dentro del contexto de este paradigma.

Modelos de UML 2.2

En UML 2.2 existen modelos estructurales y otros de comportamiento que –como sus nombres lo sugieren– se utilizan para modelar aspectos estructurales y de comportamiento, respectivamente, de las aplicaciones de software.

Los modelos estáticos o estructurales sirven para modelar el conjunto de objetos, clases, relaciones y sus agrupaciones, presentes en un sistema. Por ejemplo, una empresa tiene clientes, proveedores, empleados; los empleados, que tienen una documentación y un sueldo, se asignan a proyectos; los proyectos pueden ser internos o externos; los segundos tienen clientes, mientras que los primeros, no; los proyectos externos tienen costo y precio de venta, mientras que los internos solamente costo, etc.

Pero, además, existen cuestiones dinámicas o de comportamiento que definen cómo evolucionan esos objetos a lo largo del tiempo, y cuáles son las causas de esa evolución. Por ejemplo, un empleado puede pasar de un proyecto a otro; el sueldo de un empleado puede variar al recibir un bono anual; un proyecto puede pasar del estado de aprobado al de comenzado, o del de terminado al de aceptado; la preventa de un proyecto puede necesitar de ciertas actividades definidas en un flujo.

UML sirve para definir ambos tipos de modelos. Esto es interesante por la interrelación substancial entre ambas cuestiones. Aquí radica también su carácter de “unificado”, ya que otras notaciones previas se centraban sólo en aspectos estructurales (como los diagramas de entidades y relaciones, o DER), o sólo en aspectos de comportamiento (como las redes de Petri).

UML trabaja con 13 tipos de diagramas.

Los diagramas estructurales o estáticos de UML 2.2 son:

- Diagrama de casos de uso.
- Diagrama de objetos (estático).
- Diagrama de clases.
- Diagrama de paquetes.
- Diagrama de componentes.
- Diagrama de despliegue.
- Diagrama de estructuras compuestas.

Y los diagramas de comportamiento o dinámicos son:

- Diagrama de secuencia.
- Diagrama de comunicación (o de colaboración).
- Diagrama de máquina de estados o de estados.
- Diagrama de actividades.
- Diagrama de visión global de la interacción.
- Diagrama de tiempos

3. FORMULACIÓN DE LAS ACTIVIDADES DE APRENDIZAJE

- Materiales:
  Lectura
  Portátil ó Computador de Escritorio
- Ambiente Requerido
- Descripción de la(s) Actividad(es)

MODELADO DE REQUISITOS

Una vez que se hayan levantado los requerimientos del sistema, ya sea de forma tradicional o de forma ágil, se procede a crear los diagramas UML

CASOS DE USO

Los casos de uso son herramientas de modelización de requisitos funcionales, que preceden (en el tiempo) y exceden (en alcance) a UML. Pero fueron UML y UP los que les dieron mayor difusión. Tal vez por esa asociación con UP, los métodos ágiles evitan hablar de casos de uso.

Un caso de uso especifica una interacción entre un actor y el sistema, de modo tal que pueda ser entendida por una persona sin conocimientos técnicos. Es importante también que capte una función visible para un actor. Es posible que sirva de contrato entre el equipo de desarrollo y los interesados en el mismo.

Los actores son los roles de los agentes externos que necesitan algo del sistema. Pueden ser personas o no: por ejemplo, un actor puede ser otra aplicación que se comunica con la nuestra para solicitar algún servicio. Pusimos también el énfasis en destacar que son roles y no personas con nombre y apellido. Por ejemplo, el empleado de una empresa cliente de SenaGestiónAsistencia puede ser, a la vez, administrador y usuario de reportes, pero como actores se trata de dos roles diferentes.

Utilizando User Stories

Hay métodos de desarrollo que no utilizan casos de uso, destacándose los user stories de XP y la mayor parte de los métodos ágiles. En realidad, una user story es un requisito expresado de manera simple y en términos del usuario, por esta razón podría ser muy similar a un caso de uso de poco detalle. Lo único que las diferencia de éstos es que no hay tanta formalidad en su descripción. Habitualmente, una user story se expresa con una oración en los siguientes términos:

```
    Como administrador

    Quiero que el sistema permita dar de alta una nueva
empresa cliente, incluyendo al administrador del cliente,
más su usuario y clave

    Para permitirle el uso del sistema
```

La idea de las user stories es que, al definir solamente requisitos de alto nivel, sirven para tener una visión global del alcance y de los beneficios esperados. Además, sirven para hacer estimaciones gruesas, planificaciones y seguimiento de los proyectos de desarrollo.

Como una user story no alcanza para precisar en detalle un requisito, a menudo se la acompaña con pruebas de aceptación del usuario (user acceptance test o UAT).

A los efectos del resto de las cuestiones, cuando hablemos de casos de uso, se puede reemplazar este término por el de user story, tal vez acompañada de sus UAT.

Los escenarios que tendría esta user stories seria:

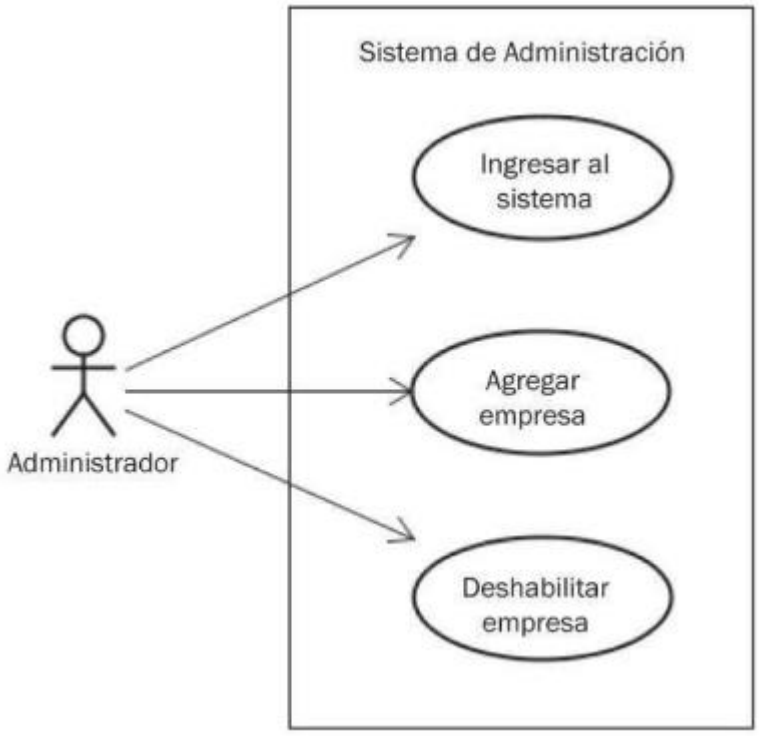
```
    El usuario solicita dar de alta una nueva empresa

    El sistema muestra los datos a ser ingresados:

           Nombre (*)

           Domicilio

           Nombre del administrador del cliente (*)

           Mail del administrador del cliente (*)

           Teléfono de contacto (*)

    El usuario completa los campos:

           Nombre: "Desarrolladores del Sur SA"

           Domicilio: "Av. Boyacá 22345 - Esquel - Chubut"

           Nombre del administrador del cliente: "Juan Pérez"

           Mail del administrador del cliente: "jperez@dscom"

           Teléfono de contacto: "02945-112564"

    El sistema valida los datos y muestra un error en el
formato del mail

    El usuario abandona la operación

    El sistema no cambia la base de datos ni genera un
usuario y una clave para el administrador del cliente.

    El sistema guarda en el log "El usuario Administrador
abandonó el alta de una empresa".
```

DIAGRAMAS DE CASOS DE USO

El modelo de casos de uso suele servir, entre otras cosas, para delimitar el alcance del sistema, esbozar quiénes interactuarán con el sistema, a modo de actores, cuáles son las funcionalidades esperadas y capturar un primer glosario de términos del dominio. Y, por sobre todas las cosas, para validar los requisitos con el cliente.

UML, como notación de modelado visual, define un tipo de diagrama denominado de casos de uso. Estos diagramas no especifican el comportamiento de los casos de uso, sino solamente relaciones entre distintos casos de uso, y de casos de uso con actores. Por eso en nuestra clasificación los hemos incluido entre los diagramas estructurales y no de comportamiento, aun cuando no ignoremos que un caso de uso, especificado con todos sus detalles, es un modelo de comportamiento.

Cuando al diagrama de casos de uso se lo va a utilizar para describir el contexto de un sistema o subsistema, se suele rodear los casos de uso por un rectángulo que denote la frontera del sistema o subsistema.



Representación de un diagrama de uso basado en la user story.

DIAGRAMA DE SECUENCIA DEL SISTEMA

Hay quienes no usan diagramas de actividades para modelar el comportamiento de los casos de uso, y prefieren, en cambio, los diagramas de secuencia de sistema.

Los diagramas de secuencia de un sistema son diagramas en los cuales se colocan dos entidades, el Actor y el Sistema, y se modelan las actividades con el paso de mensajes entre ambos.

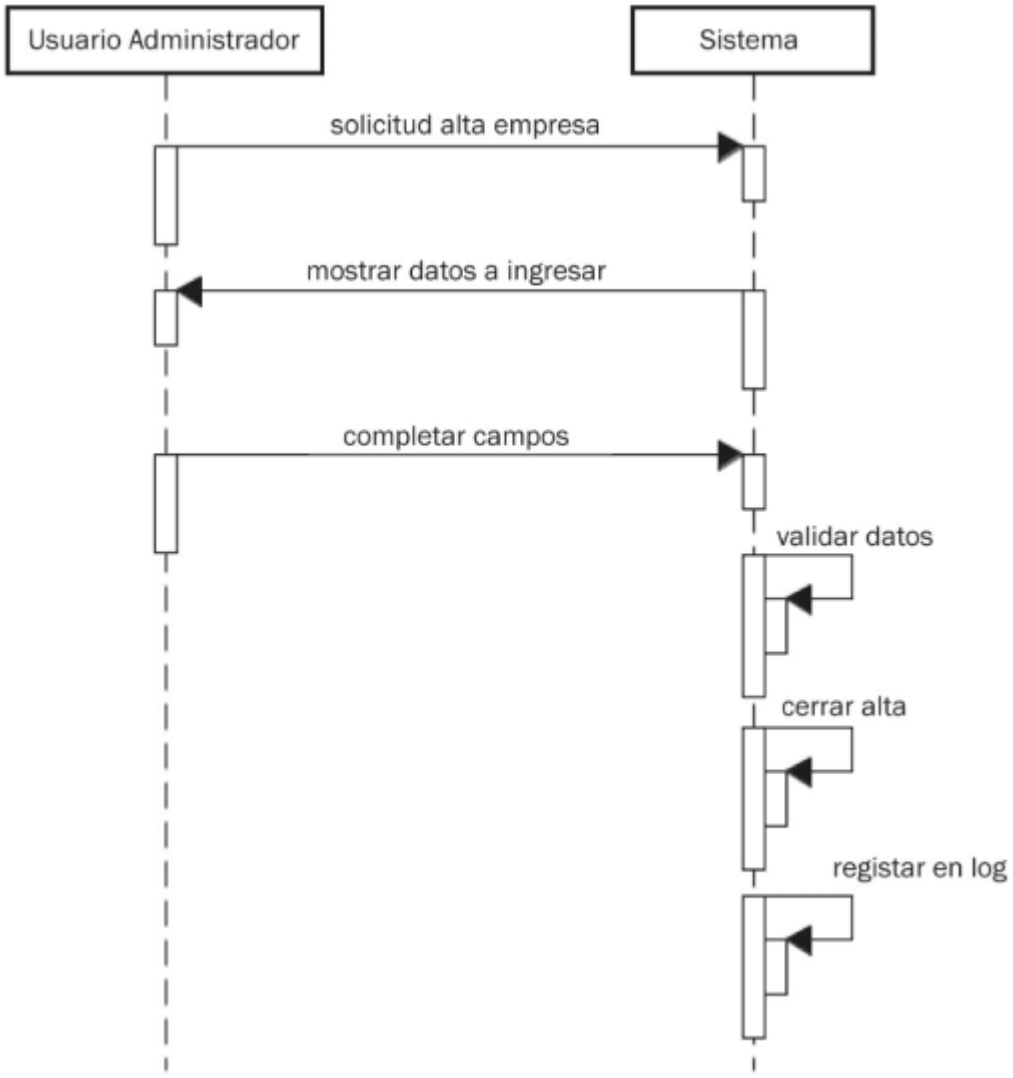


Diagrama de secuencia de la user story (Revisar los escenarios)

ANÁLISIS ORIENTADO A OBJETOS

La orientación a objetos es un paradigma que, entre otras cosas, busca reducir la brecha entre el dominio del problema y el dominio de la solución, como camino para construir software más complejo con mayor simplicidad. Por eso nos hemos centrado en actividades destinadas a definir el sistema que se construirá: el modelado del dominio del problema lo vimos en el capítulo de requisitos, mientras que en éste nos vamos a concentrar en definir la solución.

Para lograr este objetivo, la orientación a objetos plantea el viejo principio de modularización, o “divide y vencerás”, que consiste en dividir el problema en partes.

MODELADO DE OBJETOS Y CLASES

El concepto central de la orientación a objetos, como no podía ser de otra manera, es el de objeto.

En el paradigma de objetos, un sistema de software es siempre un conjunto de objetos que se envían mensajes y los responden. Por lo tanto, un objeto es toda entidad capaz de entender un mensaje y responder a él con algún comportamiento establecido. Además, los objetos tienen datos internos, que conforman su estado, y que les permiten responder los mensajes de maneras distintas según ese estado.

Todo objeto pertenece a una clase o, como decimos habitualmente, es instancia de una clase. En ese sentido, una clase es un conjunto de objetos. Pero, además, es la clase la que define qué mensajes puede entender un objeto y el comportamiento esperado como reacción a cada mensaje recibido. Asimismo, la clase es un molde del estado de los objetos, ya que en ella está definida la estructura interna del objeto.

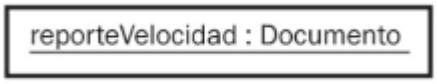
Por lo tanto, los conceptos de objeto y clase están muy asociados. Si bien los objetos son las únicas entidades que tienen existencia en un programa orientado a objetos, las clases son las que definen cuáles son los estados y comportamientos posibles de los objetos que son sus instancias.

Una clase, en definitiva, es un conjunto de objetos con la misma estructura y con el mismo comportamiento. La relación entre objeto y clase se denomina instanciación. Cada objeto particular es una instancia de la clase.

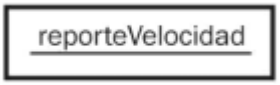
UML, que es una notación orientada a objetos, permite modelar los dos conceptos.

Modelado simple de objetos

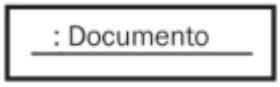
Un objeto se representa con un rectángulo en el que figura el nombre del objeto y la clase de la que éste es instancia, subrayados.



Hay ocasiones en las que necesitamos modelar un objeto sin que nos interese conocer exactamente su clase.



Otras en las que nos interesa representar una instancia de una clase sin darle un nombre.



Los objetos, en un sistema, pueden relacionarse con otros objetos. Esto se puede representar con un diagrama de objetos que muestre varios de ellos en un momento dado, con sus relaciones o enlaces.

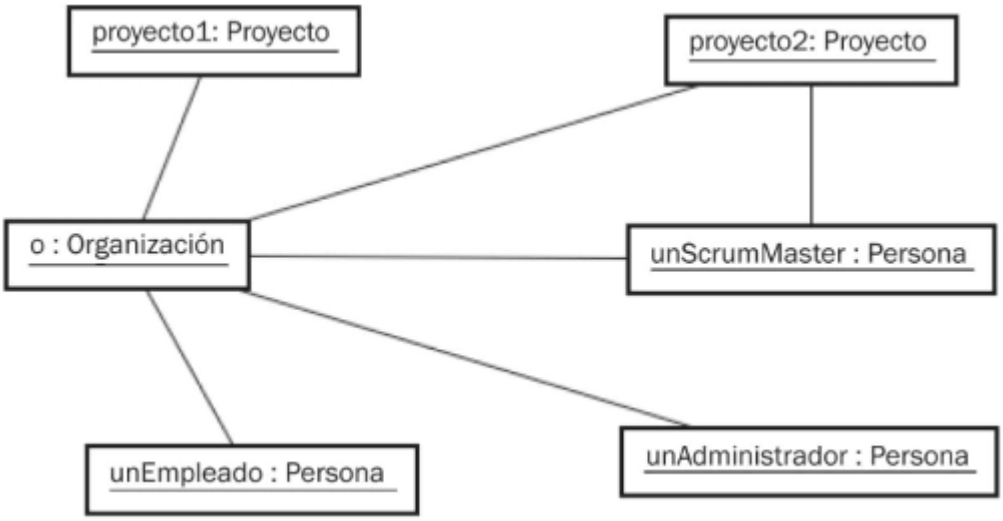


Diagrama de objetos

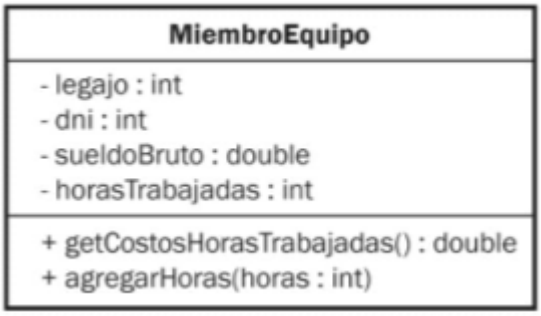
DIAGRAMA DE CLASES

La clase es una construcción de todos los lenguajes orientados a objetos. Esto hace que el diagrama de clases sea el diagrama estructural más importante a la hora de modelar diseño detallado y programación.

El diagrama de clases se puede representar en asociaciones, agregación, composición, generalización, dependencia, atributos y métodos. Algunas veces se usan también en el diseño detallado. Otras no tanto, como los nombres de asociaciones.

Elementos básicos en diagramas de clases

Hay algunas cuestiones de los diagramas de clases que pueden estar trabajando en un nivel mayor de abstracción. Los tipos de métodos, parámetros y atributos y la visibilidad de atributos y métodos.



En la anterior figura podemos destacar:

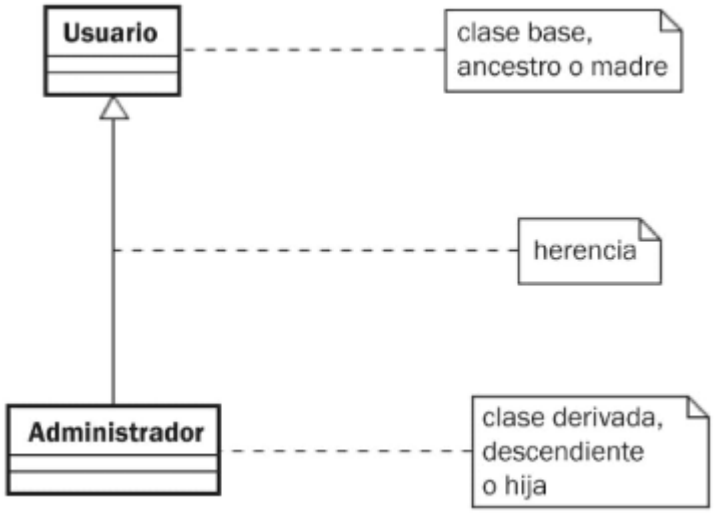
- Se puede indicar el tipo de un atributo. Por ejemplo, el atributo legajo es de tipo int, mientras que sueldoBruto es de tipo double. Esto sólo tiene sentido en lenguajes de programación en los que las variables se definan con tipos que luego no se pueden cambiar.

- Se puede especificar el tipo del valor que devuelve un método. Así lo hicimos con el método getCostoHorasTrabajadas, al decir que devuelve un valor de tipo double. El método agregarHoras, en cambio, como no le ponemos el tipo del valor devuelto, asumimos que no devuelve ningún valor.
- También podemos indicar una lista de parámetros para los métodos, y colocarles los tipos. Así, el método agregarHoras tiene el parámetro horasde tipo int.
- Es posible indicar la visibilidad de un atributo o método con los calificadores de visibilidad +, -, # y ~ .Los calificadores de visibilidad tienen los siguientes significados

CALIFICADOR	SIGNIFICADO
+	Visibilidad pública. Habitualmente significa la misma visibilidad que la clase que contiene el elemento. Esto es: el elemento es visible para cualquier otro elemento para el que la clase sea visible.
-	Visibilidad privada. Habitualmente significa que el elemento sólo es visible dentro de la clase que lo contiene. En algunos lenguajes, como ocurre en Smalltalk, sólo es visible para el objeto en sí mismo, no para la clase.
#	Visibilidad protegida. Habitualmente significa que el elemento sólo es visible dentro de la clase que lo contiene y las clases que heredan de ella.
~	Visibilidad de paquete. Especialmente introducido en UML para representar la visibilidad de paquete de Java. Habitualmente significa que el elemento sólo es visible para las clases del mismo paquete de la clase que lo contiene.

No obstante, los distintos lenguajes pueden definir significados distintos para las visibilidades, y lo usual es que, en un diagrama de diseño, la visibilidad indique el mismo significado que en el lenguaje de programación en el que estemos trabajando. La expresión “habitualmente”, es para no hacer foco en ninguna particularidad de los lenguajes de programación.

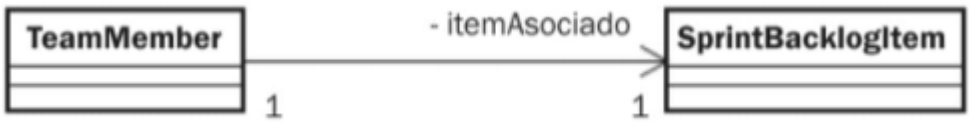
Hay lenguajes que necesitan que las instancias de las clases se creen con un método especial, habitualmente denominado constructor. Aun cuando así no fuera, hay métodos que tienen como misión principal la creación de una instancia. A estos métodos se los puede modelar con el agregado del estereotipo <<create>>.
La especialización-generalización suele estar implementada en los lenguajes orientados a objetos, mediante un mecanismo denominado herencia.



En el anterior diagrama, la clase Usuario generaliza a la clase Administrador, o lo que es lo mismo, la clase Administrador especializa a la clase Usuario. Entonces, decimos que la clase Administrador hereda de la clase Usuario. Hay otros términos también habituales: decimos que la clase Administrador deriva de la clase Usuario, o que desciende de ella. A la inversa, decimos que la clase Usuario es ancestro o base de la clase Administrado

Asociaciones en lenguajes de programación

Hemos visto los requisitos de distintas asociaciones en diagramas de clases. Ahora bien, ¿qué significa una asociación y sus distintos elementos en el contexto del diseño detallado?



Una asociación entre dos clases, en el nivel de implementación, significa que una clase tiene un atributo cuyo tipo es otra clase. La navegabilidad nos indica cuál es la clase del atributo, y el nombre del atributo es el rol de la asociación en el sentido de la navegabilidad.

En cuanto a la multiplicidad de las asociaciones, el valor 1 es el único que no ofrece matices. Cualquier valor mayor o rango de valores, implica alguna forma de colección de objetos como atributo de la clase. Si el rango de valores tuviera una cota superior o inferior, esto es, una indicación para que el programador la tenga en cuenta en el momento de la codificación.

Hay quienes en vez de una asociación calificada usan una clase de asociación, como la que vimos, con otro fin.

La anterior figura es un ejemplo de esto. Sin embargo, lo más usual es usar asociaciones calificadas en estos casos, dejando las clases de asociación para los diagramas más conceptuales.

Para revisar un ejemplo práctico con los diagramas de clases, revisa el siguiente link

4. ACTIVIDADES DE EVALUACIÓN

Estas son las actividades que se deben entregar en esta guía:

Evidencia de Conocimiento: **Pregunta- respuesta** Evaluación online en clase sobre los conceptos de diagramas UML

Evidencia de Desempeño: Envío de los **diagramas de casos de uso, diagrama de secuencia, diagrama de objetos y diagrama de clases** de su proyecto.

Evidencia de Producto: Realizar un documento donde se evidencia la construcción de **los diagramas requeridos**.

Evidencias de Aprendizaje	Criterios de Evaluación	Técnicas e Instrumentos de Evaluación
Evidencias de Conocimiento:	Reconoce los principales diagramas utilizados en desarrollo de software	Cuestionario
Evidencias de Desempeño:	Reconocer las diferencias entre los diagramas UML vistos en l guía.	Presentación y respuestas.
Evidencias de Producto:	Entrega de la las actividades planteadas a entregar.	Revisión de archivos.

1. **GLOSARIO DE TÉRMINOS**
   De acuerdo a la práctica realizar su propio glosario de términos.

6. REFERENTES BIBLIOGRÁFICOS

UML Modelado de Software para Profesionales, Fontela

Construya o cite documentos de apoyo para el desarrollo de la guía, según lo establecido en la guía de desarrollo curricular.

7. CONTROL DEL DOCUMENTO

	Nombre	Cargo	Dependencia	Fecha
Autor (es)	**Néstor Rodríguez**	**Instructor**	**Teleinformática**	**Septiembre-2019**

8. CONTROL DE CAMBIOS (diligenciar únicamente si realiza ajustes a la guía)

	Nombre	Cargo	Dependencia	Fecha	Razón del Cambio
Autor (es)					