
ANALYZE
DEVELOPMENT
DESIGN
IMPLEMENT

CASO PRÁCTICO

En la empresa siguen trabajando en diferentes aplicaciones con un nivel alto de complejidad, se desarrolla para diferentes plataformas, en entornos de ventanas, para la web, dispositivos móviles, etc. Ada lleva un tiempo observando a su equipo, y a pesar de que ya han hablado de las diferentes fases de desarrollo del software, y que están descubriendo nuevos entornos de programación que han facilitado su trabajo enormemente, se ha dado cuenta de que todavía hay una asignatura pendiente, sus empleados no utilizan herramientas ni crean documentos en las fases previas del desarrollo de una aplicación, a pesar de ser algo tan importante como el resto de fases del proceso de elaboración de software. Tampoco construyen modelos que ayuden a hacerse una idea de cómo resultará el proyecto. Estos documentos y modelos son muy útiles para que todo el mundo se ponga de acuerdo en lo que hay que hacer, y cómo van a hacerlo.

Como Ada muy bien conoce, un proyecto de software tendrá éxito sólo si produce un software de calidad, consistente y sobre todo que satisfaga las necesidades de los usuarios que van a utilizar el producto resultante.

Para desarrollar software de calidad duradera, hay que idear una sólida base arquitectónica que sea flexible al cambio.

Incluso para producir software de sistemas pequeños sería bueno hacer análisis y modelado ya que redunda en la calidad, pero lo que sí es cierto, es que cuanto más grande y complejos son los sistemas más importantes es hacer un buen modelado ya que nos ayudará a entender el comportamiento del sistema en su totalidad. Y cuando se trata de sistemas complejos el modelado nos dará una idea de los recursos necesarios (tanto humanos como materiales) para abordar el proyecto. También nos dará una visión más amplia de cómo abordar el problema para darle la mejor solución.

Ada se da cuenta de que el equipo necesita conocer procedimientos de análisis y diseño de software, así como alguna herramienta que permita generar los modelos y la documentación asociada, así que decide reunir a su equipo para empezar a tratar este tema...

1. Introducción a la orientación a objetos.

La construcción de software es un proceso cuyo objetivo es dar solución a problemas utilizando una herramienta informática y tiene como resultado la construcción de un programa informático. Como en cualquier otra disciplina en la que se obtenga un producto final de cierta complejidad, si queremos obtener un producto de calidad, es preciso realizar un proceso previo de análisis.

Enfoque orientado a objetos.

La orientación a objetos ha roto con esta forma de hacer las cosas. Con este nuevo paradigma (*modelo o patrón aplicado a cualquier disciplina científica u otro contexto epistemológico*) el proceso se centra en simular los elementos de la realidad asociada al problema de la forma más cercana posible. La abstracción (*aislar un elemento de su contexto o del resto de elementos que le acompañan para disponer de ciertas características que necesitamos excluyendo las no pertinentes. Con ello capturamos algo en común entre las diferentes instancias con objeto de controlar la complejidad del software*) que permite representar estos elementos se denomina objeto, y tiene las siguientes características:

Un **objeto** está formado por un conjunto de **atributos**, que son los datos que le caracterizan y un conjunto de **operaciones** que definen su comportamiento. Las operaciones asociadas a un objeto actúan sobre sus atributos para modificar su estado. Cuando se indica a un objeto que ejecute una operación determinada se dice que se le pasa un **mensaje**.

Las **aplicaciones orientadas a objetos están formadas por un conjunto de objetos que interaccionan enviándose mensajes para producir resultados.** Los objetos similares se abstraen en clases, se dice que un **objeto** es una **instancia** de una **clase**.

Cuando se **ejecuta** un **programa orientado** a **objetos** ocurren tres sucesos:

- ✓ Primero, los objetos se crean a medida que se necesitan.
- ✓ Segundo. Los mensajes se mueven de un objeto a otro (o del usuario a un objeto) a medida que el programa procesa información o responde a la entrada del usuario.
- ✓ Tercero, cuando los objetos ya no se necesitan, se borran y se libera la memoria.

Todo acerca del mundo de la orientación a objetos se encuentra en la página oficial del *Grupo de gestión de objetos*:

http://www.omg.org/index.htm

2. Conceptos de Orientación a Objetos.

Como hemos visto la orientación a objetos trata de acercarse al contexto del problema lo más posible por medio de la simulación de los elementos que intervienen en su resolución y basa su desarrollo en los siguientes conceptos:

- ✓ **Abstracción**: Permite capturar las características y comportamientos similares de un conjunto de objetos con el objetivo de darles una descripción formal. La abstracción es clave en el proceso de análisis y diseño orientado a objetos, ya que mediante ella podemos llegar a armar un conjunto de clases que permitan modelar la realidad, o el problema que se quiere atacar.
- ✓ **Encapsulación**: Significa reunir todos los elementos que pueden considerarse pertenecientes a una misma entidad, al mismo nivel de abstracción. Esto permite aumentar la cohesión de los componentes del sistema. Algunos autores confunden este concepto con el principio de ocultación, principalmente porque se suelen emplear conjuntamente.
- ✓ **Modularidad**: Propiedad que permite subdividir una aplicación en partes más pequeñas (llamadas módulos), cada una de las cuales debe ser tan independiente como sea posible de la aplicación en sí y de las restantes partes. En orientación a objetos es algo consustancial, ya que los objetos se pueden considerar los módulos más básicos del sistema.
- ✓ **Principio de ocultación**: Aísla las propiedades de un objeto contra su modificación por quien no tenga derecho a acceder a ellas. Reduce la propagación de efectos colaterales cuando se producen cambios.
- ✓ **Polimorfismo**: Consiste en reunir bajo el mismo nombre comportamientos diferentes. La selección de uno u otro depende del objeto que lo ejecute.
- ✓ **Herencia**: Relación que se establece entre objetos en los que unos utilizan las propiedades y comportamientos de otros formando una jerarquía. Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen.
- ✓ **Recolección de basura**: Técnica por la cual el entorno de objetos se encarga de destruir automáticamente los objetos, y por tanto desvincular su memoria asociada, que hayan quedado sin ninguna referencia a ellos.

2.1. Ventajas de la orientación a objetos.

Este paradigma tiene las siguientes **ventajas** con respecto a otros:

1. **Permite desarrollar** software en mucho **menos tiempo**, con **menos coste** y de **mayor calidad** gracias a la **reutilización** (*utilizar artefactos existentes durante la construcción de nuevo software. Esto aporta calidad y seguridad al proyecto, ya que el código reutilizado ya ha sido probado*) porque al ser completamente modular facilita la creación de código reusable dando la posibilidad de reutilizar parte del código para el desarrollo de una aplicación similar.

2. Se consigue **aumentar** la **calidad** de los **sistemas**, haciéndolos más **extensibles** (*principio de diseño en el desarrollo de sistemas informáticos que tiene en cuenta el futuro crecimiento del sistema. Mide la capacidad de extender un sistema y el esfuerzo necesario para conseguirlo*) ya que es muy sencillo aumentar o modificar la funcionalidad de la aplicación modificando las operaciones.

3. El software orientado a objetos es más **fácil de modificar y mantener** porque se basa en criterios de modularidad y encapsulación en el que el sistema se descompone en objetos con unas responsabilidades claramente especificadas e independientes del resto.

4. La tecnología de objetos **facilita** la **adaptación** al **entorno** y el **cambio** haciendo **aplicaciones escalables** (*propiedad deseable de un sistema, red o proceso que le permite hacerse más grande sin rehacer su diseño y sin disminuir su rendimiento*). Es sencillo modificar la estructura y el comportamiento de los objetos sin tener que cambiar la aplicación.

2.2. Clases, atributos y métodos.

Los objetos de un sistema se abstraen, en función de sus características comunes, en clases. Una clase está formada por un conjunto de procedimientos y datos que resumen características similares de un conjunto de objetos. La clase tiene dos propósitos: definir **abstracciones** y favorecer la **modularidad**.

Una **clase** se describe por un conjunto de elementos que se denominan **miembros** y que son:

- ✓ **Nombre**.
- ✓ **Atributos**: conjunto de características asociadas a una clase. Pueden verse como una relación binaria entre una clase y cierto dominio formado por todos los posibles valores que puede tomar cada atributo. Cuando toman valores concretos dentro de su dominio definen el estado del objeto. Se definen por su nombre y su tipo, que puede ser simple o compuesto como otra clase.
- ✓ **Protocolo**: Operaciones (métodos, mensajes) que manipulan el estado. Un *método* es el procedimiento o función que se invoca para actuar sobre un objeto. Un *mensaje* es el resultado de cierta acción efectuada por un objeto. Los métodos determinan como actúan los objetos cuando reciben un mensaje, es decir, cuando se requiere que el objeto realice una acción descrita en un método se le envía un mensaje. El conjunto de mensajes a los cuales puede responder un objeto se le conoce como *protocolo del objeto*.

Por ejemplo, si tenemos un objeto icono, tendrá como atributos el tamaño, o la imagen que muestra, y su protocolo puede constar de mensajes invocados por el clic del botón de un ratón cuando el usuario pulsa sobre el icono. De esta forma los mensajes son el único conducto que conectan al objeto con el mundo exterior.

Los valores asignados a los atributos de un objeto concreto hacen a ese objeto ser único. La clase define sus características generales y su comportamiento.

2.3. Visibilidad.

El principio de ocultación es una propiedad de la orientación a objetos que consiste en aislar el estado de manera que sólo se puede cambiar mediante las operaciones definidas en una clase. Este aislamiento protege a los datos de que sean modificados por alguien que no tenga derecho a acceder a ellos, eliminando efectos secundarios e interacciones. Da lugar a que las clases se dividan en dos partes:

1. **Interfaz**: captura la visión externa de una clase, abarcando la abstracción del comportamiento común a los ejemplos de esa clase.
2. **Implementación**: comprende cómo se representa la abstracción, así como los mecanismos que conducen al comportamiento deseado.

Existen distintos niveles de ocultación que se implementan en lo que se denomina **visibilidad**. Es una característica que define el tipo de acceso que se permite a atributos y métodos y que podemos establecer como:

- ✓ **Público**: Se pueden acceder desde cualquier clase y cualquier parte del programa.
- ✓ **Privado**: Sólo se pueden acceder desde operaciones de la clase.
- ✓ **Protegido**: Sólo se pueden acceder desde operaciones de la clase o de clases derivadas (*cuando se utiliza la herencia es la clase que hereda los atributos y métodos de la clase base*) en cualquier nivel.

Como norma general a la hora de definir la visibilidad tendremos en cuenta que:

1. El estado debe ser privado. Los atributos de una clase se deben modificar mediante métodos de la clase creados a tal efecto.
2. Las operaciones que definen la funcionalidad de la clase deben ser públicas.
3. Las operaciones que ayudan a implementar parte de la funcionalidad deben ser privadas (si no se utilizan desde clases derivadas) o protegidas (si se utilizan desde clases derivadas).

3. UML.

UML (*Unified Modeling Language o Lenguaje Unificado de Modelado*) es un conjunto de herramientas que **permite modelar, construir y documentar los elementos que forman un sistema software orientado a objetos.** Se ha convertido en el estándar de facto de la industria, debido a que ha sido concebido por los autores de los tres métodos más usados de orientación a objetos: *Grady Booch*, *Ivar Jacobson* y *Jim Rumbaugh*, de hecho, las raíces técnicas de UML son:

- ✓ OMT - Object Modeling Technique (Rumbaugh et al.)
- ✓ Método-Booch (G. Booch)
- ✓ OOSE - Object-Oriented Software Engineering (I. Jacobson)

UML permite a los desarrolladores visualizar el producto de su trabajo en esquemas o diagramas estandarizados denominados modelos (*representación gráfica o esquemática de una realidad, sirve para organizar y comunicar de forma clara los elementos que involucran un todo. Esquema teórico de un sistema o de una realidad compleja que se elabora para facilitar su comprensión y el estudio de su comportamiento*) que representan el sistema desde diferentes perspectivas. Modelar tiene las siguientes ventajas:

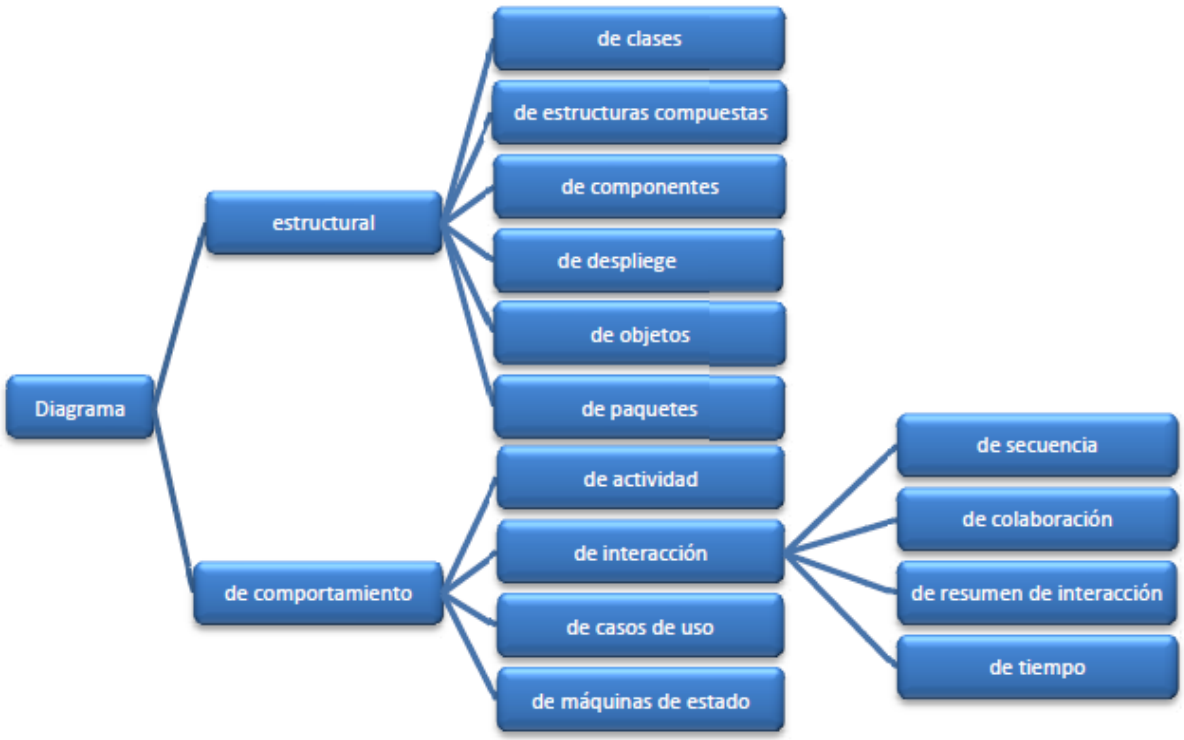
- ✓ Permite utilizar un lenguaje común que facilita la comunicación entre el equipo de desarrollo.
- ✓ Con UML podemos documentar todos los artefactos (*información que es utilizada o producida mediante un proceso de desarrollo de software. Pueden ser artefactos un modelo, una descripción o un software*) de un proceso de desarrollo (requisitos *(condiciones que debe cumplir un proyecto software. Suelen venir definidos por el cliente. Permiten definir los objetivos que debe cumplir un proyecto software)*, arquitectura, pruebas, versiones,…) por lo que se dispone de documentación que trasciende al proyecto.
- ✓ Hay estructuras que trascienden lo representable en un lenguaje de programación, como las que hacen referencia a la arquitectura del sistema (*conjunto de decisiones significativas acerca de la organización de un sistema software, la selección de los elementos estructurales a partir de los cuales se compone el sistema, y las interfaces entre ellos. Junto con su comportamiento, tal y como se especifica en las colaboraciones entre esos elementos, la composición de estos elementos estructurales y de comportamiento en subsistemas progresivamente mayores y el estilo arquitectónico que guía esta organización, estos elementos y sus interfaces, sus colaboraciones y su composición*), utilizando estas tecnologías podemos incluso indicar qué módulos de software vamos a desarrollar y sus relaciones, o en qué nodos hardware se ejecutarán cuando trabajamos con sistemas distribuidos.

- ✓ **Permite especificar todas las decisiones de análisis, diseño e implementación, construyéndose modelos precisos, no ambiguos y completos.**

3.1. Tipos de diagramas UML.

UML define un sistema como **una colección de modelos que describen sus diferentes perspectivas.** Los modelos se implementan en una serie de diagramas que son representaciones gráficas de una colección de elementos de modelado, a menudo dibujado como un grafo conexo. Se clasifican en:

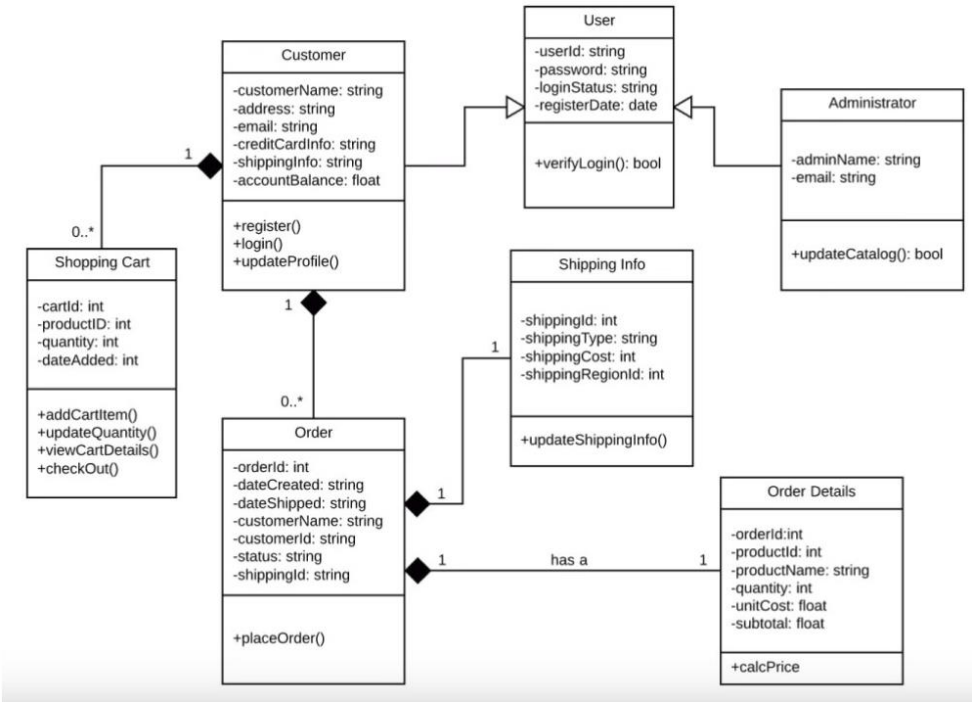
- ✓ **Diagramas Estructurales:** Representan la visión estática del sistema. Especifican clases y objetos y como se distribuyen en el sistema.
- ✓ **Diagramas de Comportamiento**: Muestran la conducta en tiempo de ejecución del sistema.



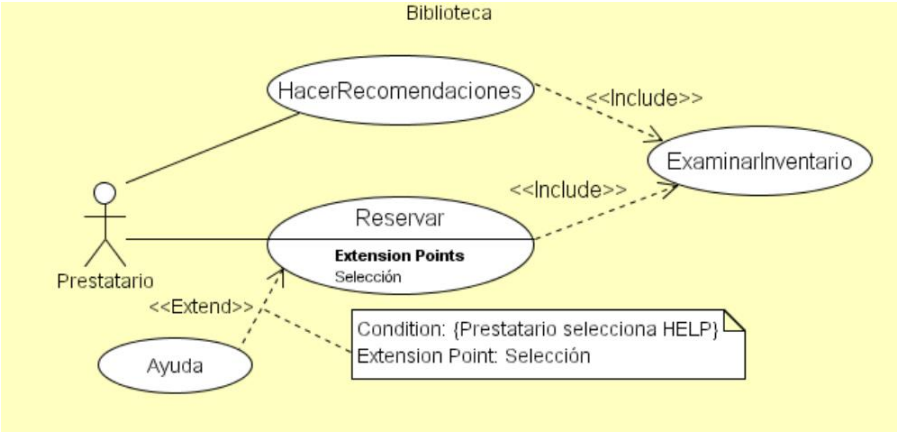
Un diagrama UML se compone de 4 tipos de elementos:

- ✓ **Estructuras**: Son los nodos del grafo y definen el tipo de diagrama.
- ✓ **Relaciones**: Son los arcos del grafo que se establecen entre los nodos del diagrama.
- ✓ **Notas**: Se representan como un cuadro donde escribiremos comentarios de ayuda a la comprensión del diagrama.
- ✓ **Agrupaciones**: Se utilizan cuando modelamos sistemas grandes para facilitar su desarrollo por bloques.

- ✓ **P.e. Diagrama de clases**



- ✓ **P.e. Diagrama de CU**



- ✓ **P.e. Diagrama de secuencias**