Tutorial de UML

Introducción:

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 lo son escribir clases en un lenguaje determinado, esquemas de base de datos y componentes de software reusables.

Objetivos:

Entregar un material de apoyo que le permita al lector poder definir diagramas propios como también poder entender el modelamiento de diagramas ya existentes.

Audiencia:

El presente documento esta orientado a alumnos que ya poseen ciertos conceptos de OOP (o están haciendo un curso) y a su vez conocen algún lenguaje Orientado a Objetos (ejemplo: C++ o Java), por lo tanto no es un curso en si, sino más bien un material de apoyo al estudiante.

Contenidos:

El documento contempla el estudio de tres diagramas:

- Modelamiento de Clases
- Casos de Uso
- Diagrama de Interacción

A su vez el estudio de un problema completo que los involucra (El Hotel), utilizando como herramienta case Rational Rose

Modelo de Clases

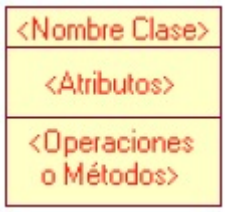
Introducción

Un diagrama de clases sirve para visualizar las relaciones entre las clases que involucran el sistema, las cuales pueden ser asociativas, de herencia, de uso y de contenimiento.
Un diagrama de clases esta compuesto por los siguientes elementos:

- Clase: atributos, métodos y 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:

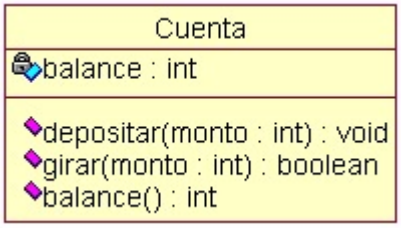


  En donde:
  - **Superior**: Contiene el nombre de la Clase
  - **Intermedio**: Contiene los atributos (o variables de instancia) que caracterizan a la Clase (pueden ser private, protected o public).
  - **Inferior**: Contiene los métodos u operaciones, los cuales son la forma como interactúa el objeto 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
  - y Balance

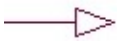
  El diseño asociado es:



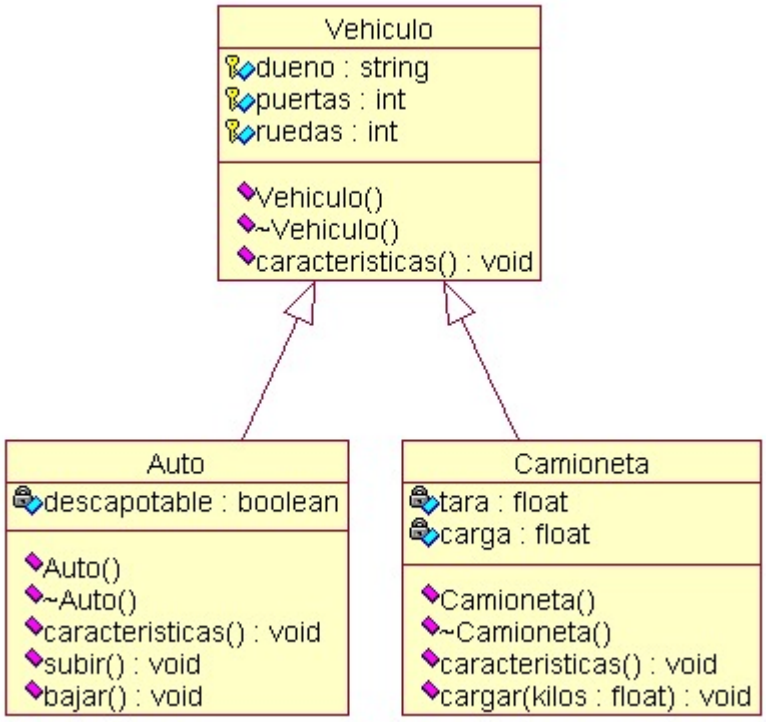
Atributos y Métodos:

- **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** (+, ): Indica que el atributo será visible tanto dentro como fuera de la clase, es decir, es accesible desde todos lados.
  - **private** (-, ): Indica que el atributo sólo será accesible desde dentro de la clase (sólo sus métodos lo pueden accesar).
  - **protected** (#, ): Indica que el atributo no será accesible desde fuera de la clase, pero si podrá ser accesado por métodos de la clase además de las subclases que se deriven (ver herencia).
- **Métodos:**
  Los métodos u operaciones de una clase son la forma en como ésta interactúa con su entorno, éstos pueden tener las características:
  - **public** (+, ): Indica que el método será visible tanto dentro como fuera de la clase, es decir, es accesible desde todos lados.
  - **private** (-, ): Indica que el método sólo será accesible desde dentro de la clase (sólo otros métodos de la clase lo pueden accesar).
  - **protected** (#, ): Indica que el método no será accesible desde fuera de la clase, pero si podrá ser accesado por métodos de la clase además de métodos de las subclases que se deriven (ver herencia).

- **Relaciones entre Clases:**
  Ahora ya definido el concepto de Clase, es necesario explicar como 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 éstas pueden ser:
  - **uno o muchos**: 1..* (1..n)
  - **0 o muchos**: 0..* (0..n)
  - **número fijo**: m (m denota el número).

iv. **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 Camioneta heredan de Vehículo, es decir, Auto posee las Características de Vehículo (Precio, VelMax, etc.) además posee algo particular que es Descapotable, en cambio Camioneta también hereda las características de Vehiculo (Precio, VelMax, etc.) pero posee como particularidad propia Tara y Carga.

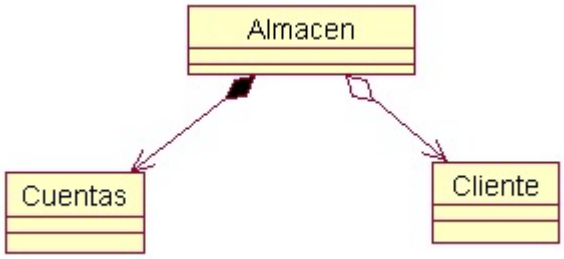
Cabe destacar que fuera de este entorno, lo único "visible" es el método Características aplicable a instancias de Vehículo, Auto y Camioneta, pues tiene definición pública, en cambio atributos como Descapotable no son visibles por ser privados.

v. **Agregación**: 

Para modelar objetos complejos, n 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:

- **Por Valor**: Es un tipo de relación estática, en donde el tiempo de vida del objeto incluido esta 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:



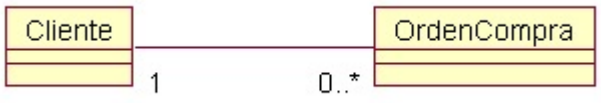
En donde se destaca que:

- 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.

vi. **Asociación**: 

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



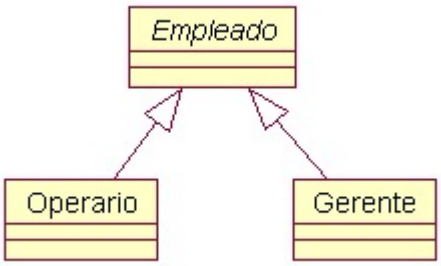
Un cliente puede tener asociadas muchas Ordenes de Compra, en cambio una orden de compra solo puede tener asociado un cliente.

vii. **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 grafica que instancia una ventana (la creación del Objeto Ventana esta condicionado a la instanciación proveniente desde el objeto Aplicación):

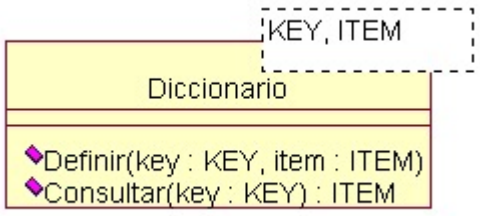
Cabe destacar que el objeto creado (en este caso la Ventana gráfica) no se almacena dentro del objeto que lo crea (en este caso la Aplicación).

- **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 (aún no han sido definidos, es decir, sin implementación). La única forma de utilizarla es definiendo subclases, que implementan los métodos abstractos definidos.

  - **Clase parametrizada**:

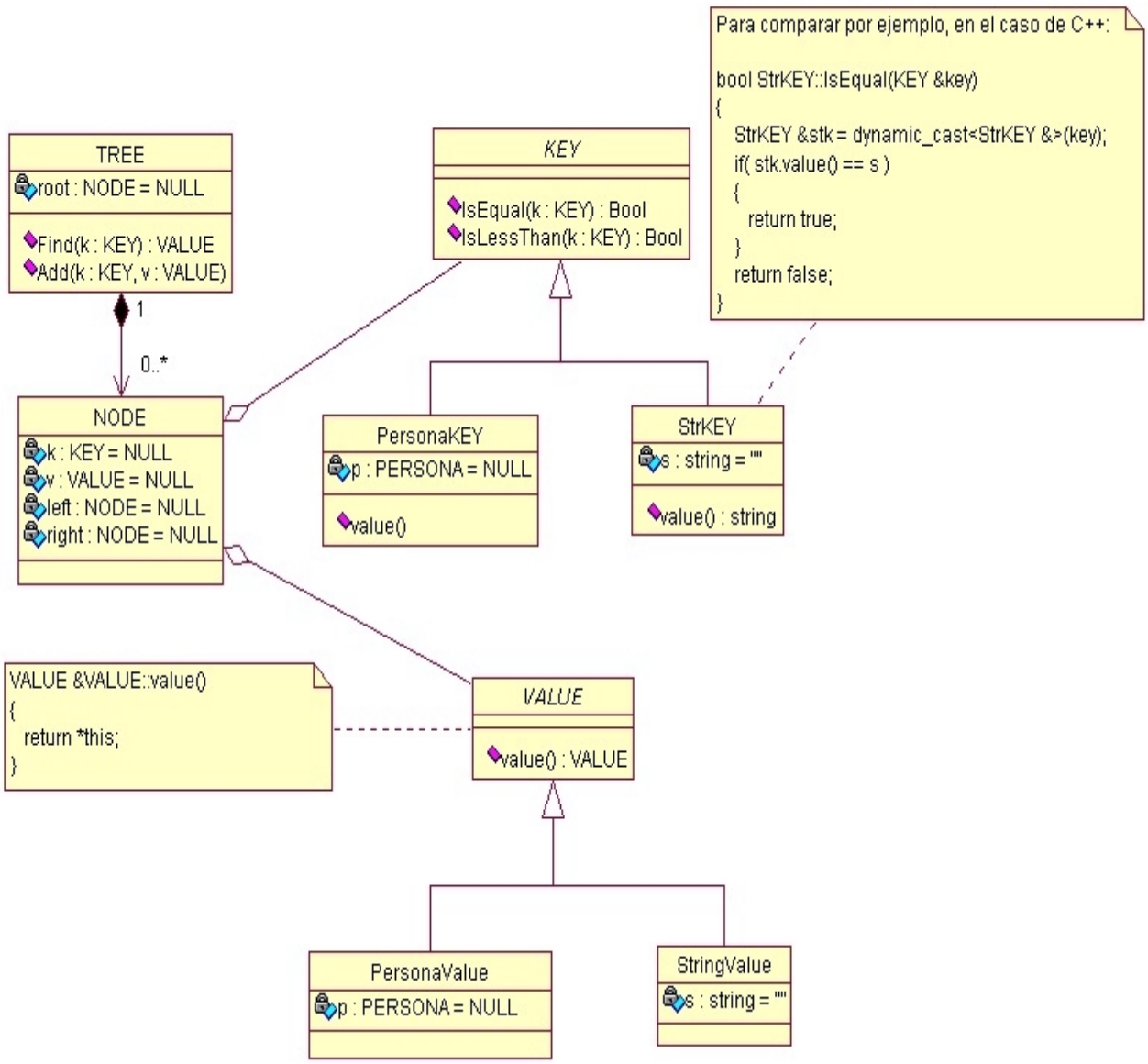


Una clase parametrizada se denota con un subcuadro en el extremo superior de la clase, en donde se especifican los parámetros que deben ser pasados a la clase para que esta pueda ser instanciada. El ejemplo más típico es el caso de un Diccionario en donde una llave o palabra tiene asociado un significado, pero en este caso las llaves y elementos pueden ser genéricos. La genericidad puede venir dada de un Témplate (como en el caso de C++) o bien de alguna estructura predefinida (especialización a través de clases).
En el ejemplo no se especificaron los atributos del Diccionario, pues ellos dependerán exclusivamente de la implementación que se le quiera dar.

Ejemplo:
Supongamos que tenemos un el caso del Diccionario implementado mediante un árbol binario, en donde cada nodo posee:

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

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


Para comparar por ejemplo, en el caso de C++:
bool StrKEY::IsEqual(KEY &key)
{
StrKEY &stk = dynamic_cast<StrKEY &>(key);
if(stk.value() == s)
{
return true;
}
return false;
}
TREE
root : NODE = NULL
Find(k : KEY) : VALUE
Add(k : KEY, v : VALUE)
1
0..*
KEY
IsEqual(k : KEY) : Bool
IsLessThan(k : KEY) : Bool
NODE
k : KEY = NULL
v : VALUE = NULL
left : NODE = NULL
right : NODE = NULL
PersonaKEY
p : PERSONA = NULL
value()
StrKEY
s : string = ""
value() : string
VALUE &VALUE::value()
{
return *this;
}
VALUE
value() : VALUE
PersonaValue
p : PERSONA = NULL
StringValue
s : string = ""

Casos de Uso (Use Case)

Introducción

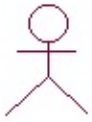
El diagrama de casos de uso representa la forma en como un Cliente (Actor) opera con el sistema en desarrollo, además de la forma, tipo y orden en como los elementos interactúan (operaciones o casos de uso).

Un diagrama de casos de uso consta de los siguientes elementos:

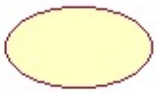
- Actor.
- Casos de Uso.
- Relaciones de Uso, Herencia y Comunicación.

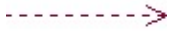
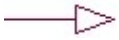
Elementos

- **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.
  - **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 esta orientado exclusivamente para casos de uso (y no para actores).
    extends: Se recomienda utilizar cuando un caso de uso es similar a otro (características).

uses: 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 esta la duda clásica de **usar** o **heredar**.

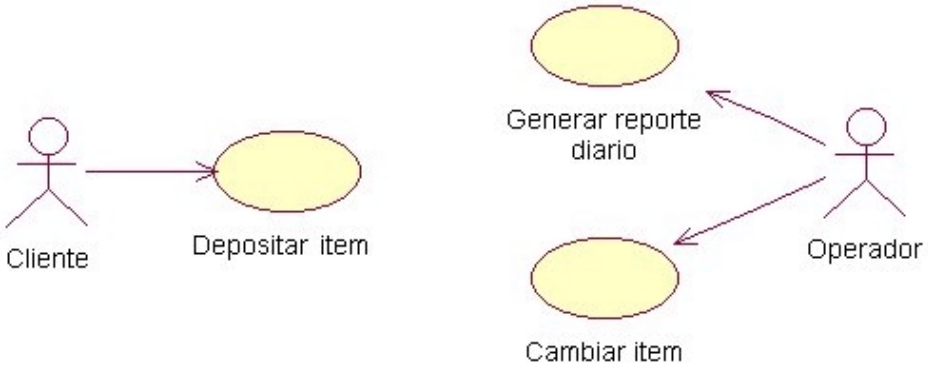
Ejemplo:

Como ejemplo esta 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:

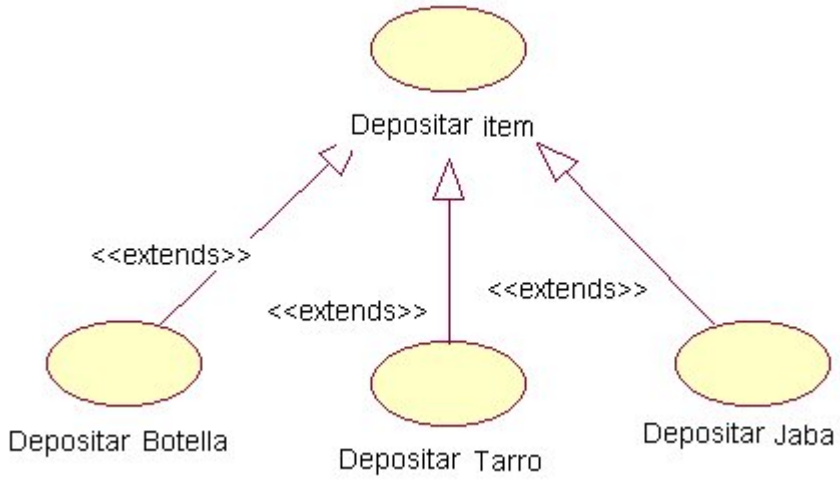
- Registrar el número de ítems ingresados.
- Imprimir un recibo cuando el usuario lo solicita:
  a. Describe lo depositado
  b. El valor de cada ítem
  c. Total
- El usuario/cliente presiona el botón de comienzo
- Existe un operador que desea saber lo siguiente:
  a. Cuantos ítems han sido retornados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

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

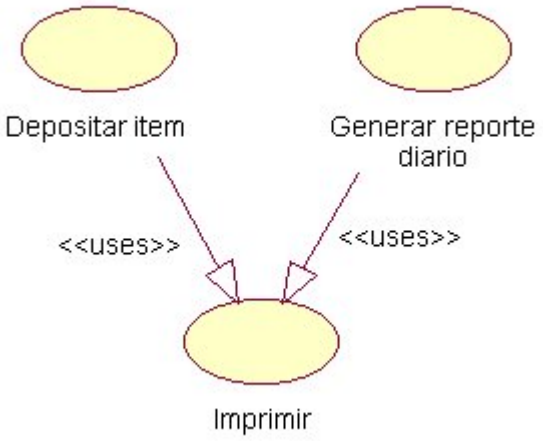
Luego, tenemos que un Cliente puede Depositar Ítems y un Operador puede cambiar la información de un Ítem o bien puede Imprimir un informe:



Además podemos 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 Use Case es:

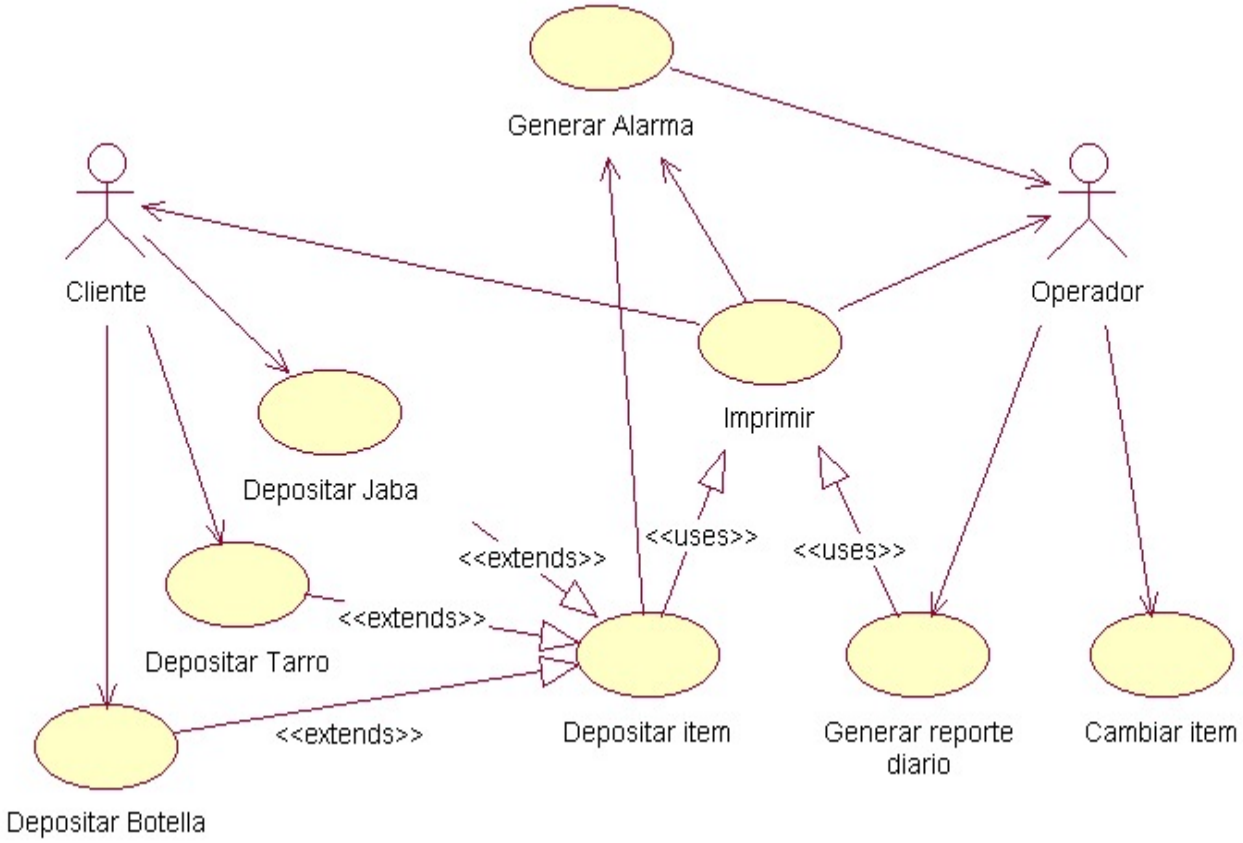

Generar Alarma
Cliente
Operador
Imprimir
Depositar Jaba
<<extends>>
<<uses>>
<<uses>>
<<extends>>
Depositar Tarro
<<extends>>
Depositar item
Generar reporte diario
Cambiar item
Depositar Botella

Diagrama de Interacción

Introducción

El diagrama de interacción, representa la forma en como un Cliente (Actor) u Objetos (Clases) se comunican entre si en petición a un evento. Esto implica recorrer toda la secuencia de llamadas, de donde se obtienen las responsabilidades claramente.

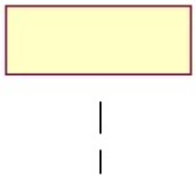
Dicho diagrama puede ser obtenido de dos partes, desde el Diagrama Estático de Clases o el de Casos de Uso (son diferentes).

Los componentes de un diagrama de interacción son:

- Un Objeto o Actor.
- Mensaje de un objeto a otro objeto.
- Mensaje de un objeto a si mismo.

Elementos

- **Objeto / Actor**:

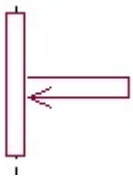


El rectángulo representa una instancia de un Objeto en particular, y la línea punteada representa las llamadas a métodos del objeto.

- **Mensaje a Otro Objeto**:

Se representa por una flecha entre un objeto y otro, representa la llamada de un método (operación) de un objeto en particular.

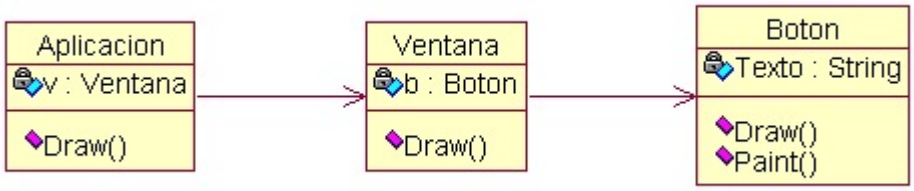
- **Mensaje al Mismo Objeto**:



No solo llamadas a métodos de objetos externos pueden realizarse, también es posible visualizar llamadas a métodos desde el mismo objeto en estudio.

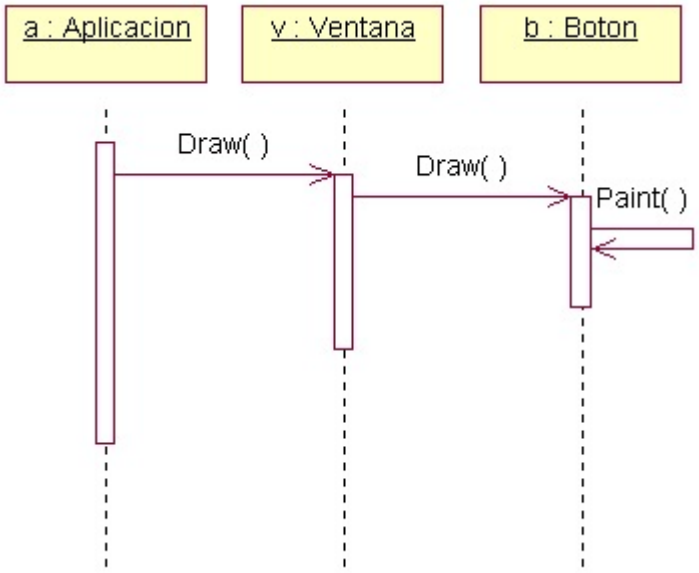
Ejemplo

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 ésta a su vez posee internamente un botón.

Entonces el diagrama de interacción para dicho modelo es:



En donde se hacen notar las sucesivas llamadas a Draw() (entre objetos) y la llamada a Paint() por el objeto Botón.

Ejemplo - Tutorial UML "Hotel"

El dueño de un hotel le pide a usted desarrollar un programa para consultar sobre las piezas disponibles y reservar piezas de su hotel.
El hotel posee tres tipos de piezas: simple, doble y matrimonial, y dos tipos de clientes: habituales y esporádicos. Una reservación almacena datos del cliente, de la pieza reservada, la fecha de comienzo y el número de días que será ocupada la pieza.
El recepcionista del hotel debe poder hacer las siguientes operaciones:

- Obtener un listado de las piezas disponible de acuerdo a su tipo
- Preguntar por el precio de una pieza de acuerdo a su tipo
- Preguntar por el descuento ofrecido a los clientes habituales
- Preguntar por el precio total para un cliente dado, especificando su numero de RUT, tipo de pieza y número de noches.
- Dibujar en pantalla la foto de un pieza de acuerdo a su tipo
- Reservar una pieza especificando el número de la pieza, rut y nombre del cliente.
- Eliminar una reserva especificando el número de la pieza

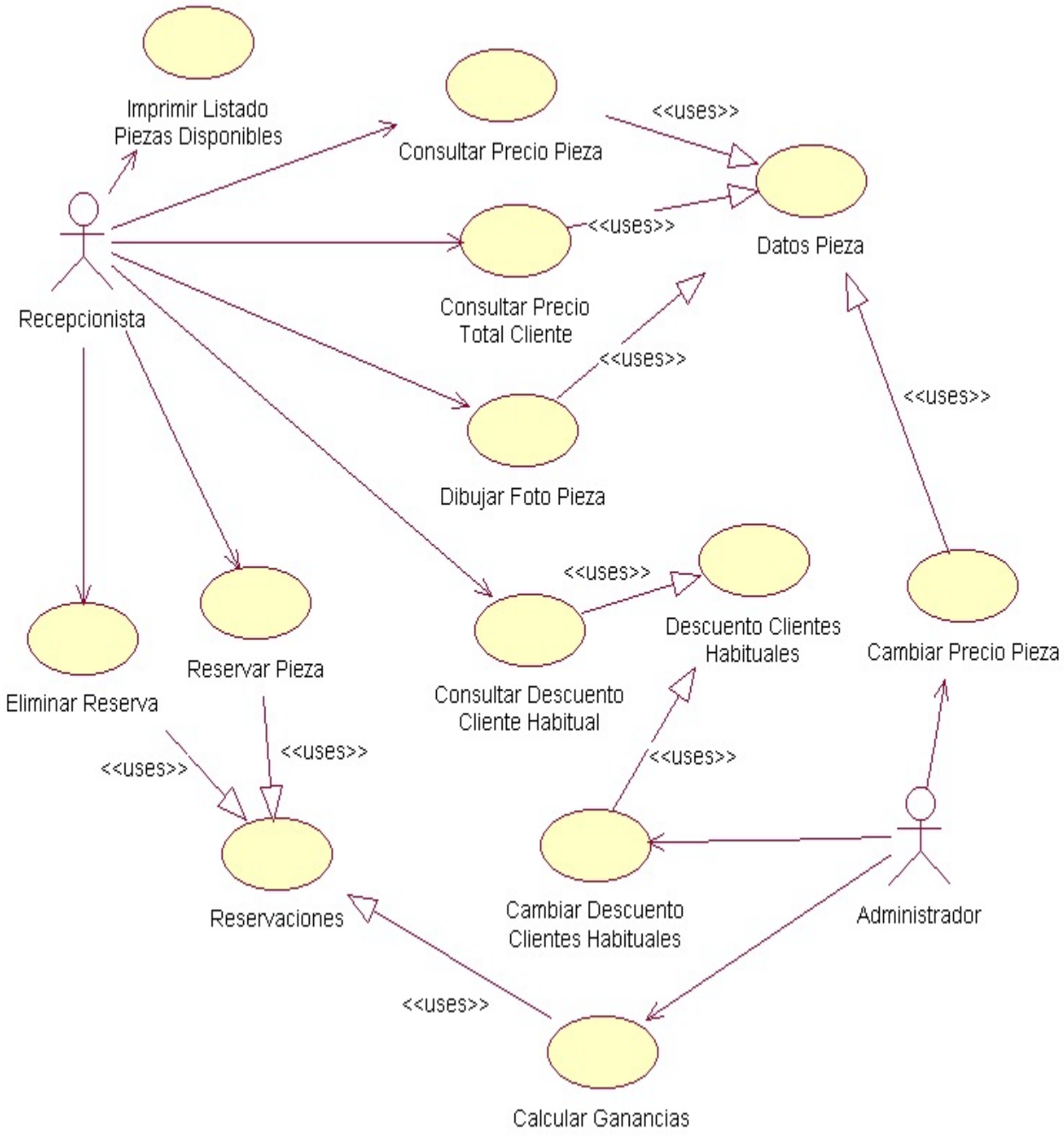
El administrador puede usar el programa para:

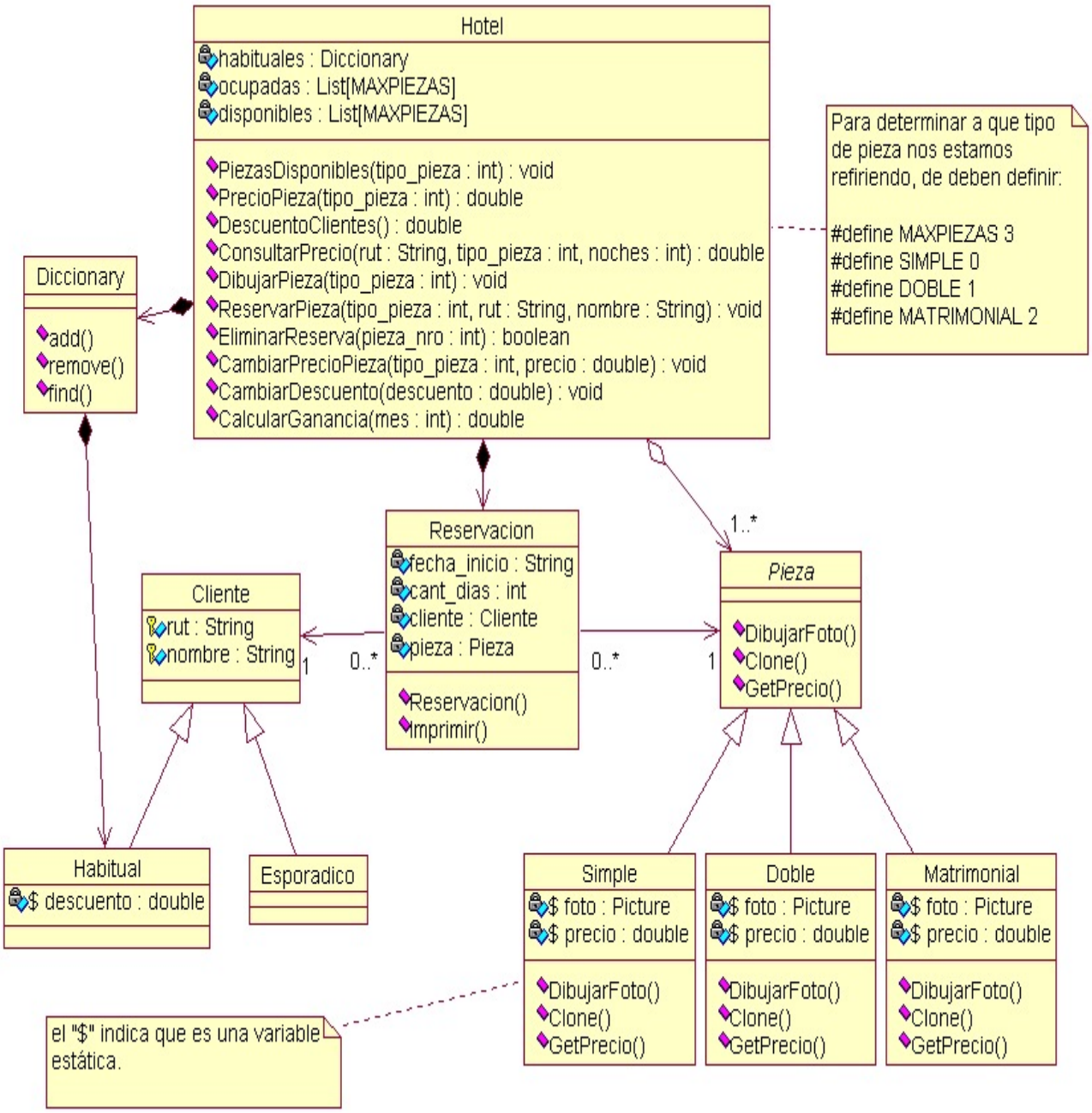
- Cambiar el precio de una pieza de acuerdo a su tipo
- Cambiar el valor del descuento ofrecido a los clientes habituales
- Calcular las ganancias que tendrán en un mes especificado (considere que todos los meses tienen treinta días).

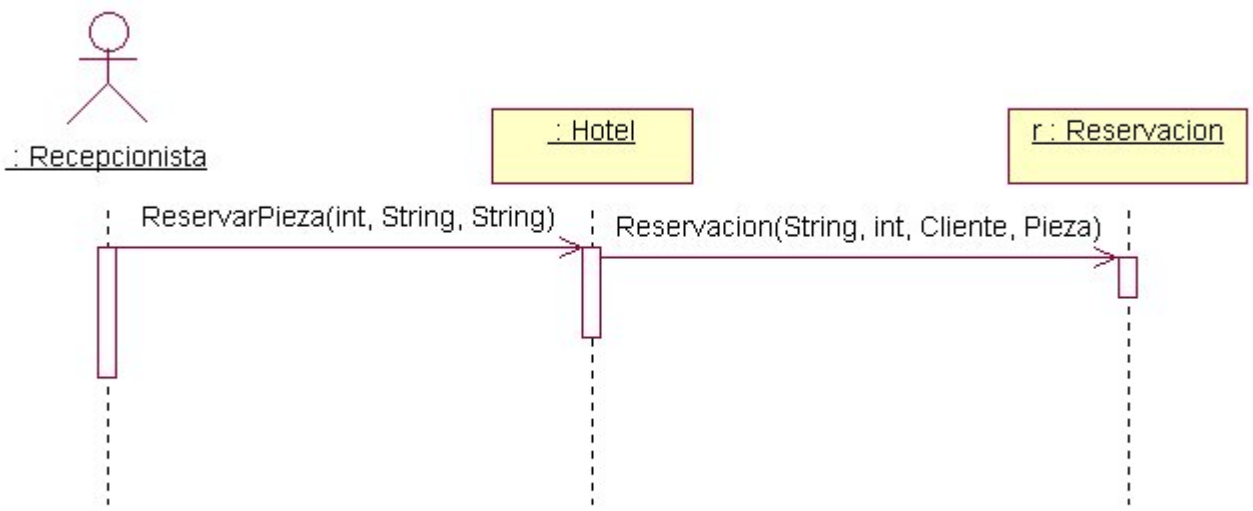
El hotel posee información sobre cuales clientes son habituales. Esta estructura puede manejarla con un diccionario, cuya clave sea el número de RUT y como significado tenga los datos personales del cliente.
El diseño a desarrollar debe facilitar la extensibilidad de nuevos tipos de pieza o clientes y a su vez permitir agregar nuevas consultas.

El presente ejemplo contempla los siguientes diagramas:

- Casos de uso
- Diagrama de clases
- Dos diagramas de interacción


Imprimir Listado Piezas Disponibles
Consultar Precio Pieza
<<uses>>
Datos Pieza
Recepcionista
Consultar Precio Total Cliente
<<uses>>
<<uses>>
Dibujar Foto Pieza
<<uses>>
Descuento Clientes Habituales
Cambiar Precio Pieza
Eliminar Reserva
Reservar Pieza
Consultar Descuento Cliente Habitual
<<uses>>
<<uses>>
<<uses>>
Reservaciones
Cambiar Descuento Clientes Habituales
Administrador
<<uses>>
Calcular Ganancias


Hotel
habituales : Diccionary
ocupadas : List[MAXPIEZAS]
disponibles : List[MAXPIEZAS]
PiezasDisponibles(tipo_pieza : int) : void
PrecioPieza(tipo_pieza : int) : double
DescuentoClientes() : double
ConsultarPrecio(rut : String, tipo_pieza : int, noches : int) : double
DibujarPieza(tipo_pieza : int) : void
ReservarPieza(tipo_pieza : int, rut : String, nombre : String) : void
EliminarReserva(pieza_nro : int) : boolean
CambiarPrecioPieza(tipo_pieza : int, precio : double) : void
CambiarDescuento(descuento : double) : void
CalcularGanancia(mes : int) : double
Para determinar a que tipo de pieza nos estamos refiriendo, de deben definir:
#define MAXPIEZAS 3
#define SIMPLE 0
#define DOBLE 1
#define MATRIMONIAL 2
Diccionary
add()
remove()
find()
Reservacion
fecha_inicio : String
cant_dias : int
cliente : Cliente
pieza : Pieza
Reservacion()
Imprimir()
1..*
Cliente
rut : String
nombre : String
1
0..*
0..*
1
Pieza
DibujarFoto()
Clone()
GetPrecio()
Habitual
$ descuento : double
Esporadico
Simple
$ foto : Picture
$ precio : double
DibujarFoto()
Clone()
GetPrecio()
Doble
$ foto : Picture
$ precio : double
DibujarFoto()
Clone()
GetPrecio()
Matrimonial
$ foto : Picture
$ precio : double
DibujarFoto()
Clone()
GetPrecio()
el "$" indica que es una variable estática.


: Recepcionista
: Hotel
r : Reservacion
ReservarPieza(int, String, String)
Reservacion(String, int, Cliente, Pieza)