{desafío} latam_

UML (Parte I)

Introducción a UML

Competencias

- Conocer el lenguaje de modelado unificado (UML).
- Conocer los distintos tipos de diagramas.
- Reconocer el paradigma de la programación en el que está basado UML.
- Entender el concepto del proceso de creación de software.

Introducción

Los tres amigos. Quizás esta poca ortodoxa forma de llamar a los creadores de una de las mayores innovaciones conceptuales de los últimos tiempos, nos parezca algo alejado a lo que estamos acostumbrados; pero la misión de la creación de tecnología tiene ese propósito. Estos "amigos" quizás no pensaron que basándose en el trabajo que realizaron mientras el último respiro del pasado siglo los acompañaba; se han creado estándares que ya son oficiales. Quizás hemos sido testigos de la aplicación de UML en varios "bocetos" abandonados en pizarras llenas de sueños, que, tal vez llegaron a ser realidad. Puede que hasta lo hayamos utilizado sin darnos cuenta, o para aquellos con más experiencia; puede que lo sigan utilizando de forma profesional. UML no solo nos crea una interfaz entre lo que se piensa y lo que se hace, si no que además, ayuda no solo a la industria de software, sino, en general, de cualquier industria que utilice modelos. Cualquiera que sea el uso que se le esté dando a UML, deben saber estos, amigos; que su trabajo sigue estando vigente, tal vez no completamente en la forma que pensaron en un principio; pero nos sigue ayudando a enfocarnos en construir en lugar de reconstruir.

Imagen 1: Comprensión de lo que de requiere desarrollar.

Algunas definiciones

¿Qué es un modelo?

Un modelo, es una representación de algo, en cierto medio (Papel, pantallas, maquetas, etc.), que capta los aspectos principales, ignorando los menos importantes o que no aporten información relevante. Un modelo en un sistema de software está expresado mediante UML.

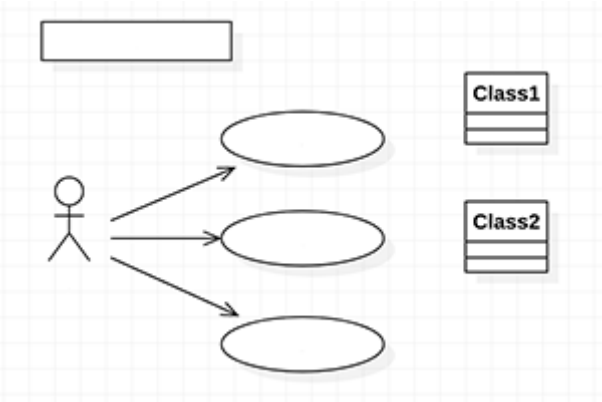

Imagen 2: Representación de un modelo.

¿Qué es UML?

UML (Unified Modeling language), es un **lenguaje** de modelado unificado, que surge a finales de la década de los 80 y principios de los 90. Entre los años 94 y 96, Grady y Jim e Ivar, conocidos como; "los tres amigos", crearon finalmente UML. Resultado de un trabajo iterativo y gradual, que pone a nuestra disposición, una norma construida sobre muchas ideas y contribuciones realizadas (unificadas), por numerosos individuos y compañías de la comunidad de la orientación a objetos.

UML es un lenguaje y no una metodología, esto suele causar confusiones. Las *metodologías* poseen un lenguaje y un proceso para modelar. En este caso; el lenguaje de modelado es la notación (gráfica), de la que se valen las metodologías para expresar los diseños.

Un poco de historia

En el año 1980, el paradigma de la orientación a objetos; ya no era parte solamente del contexto científico, en vez de ello, se encontraba ya en uso "cotidiano" en lenguajes que comenzaron a ser ampliamente utilizados como `Smalltalk`, dando paso a la creación de nuevos lenguajes basados en en la *POO*., como es el caso de `C++` y `Java`. Antes de que todo esto ocurriera, ya existían diferentes tipos de metodologías para representar el diseño de un *software* utilizando la *programación procedural*; pero ahora había que lograr el mismo éxito en la POO. Entre los años 90 y el 98, muchas fueron las organizaciones que intentaron llegar a un estándar unificado; pero finalmente "los tres amigos", y, su versión 1.1 de UML fueron los ganadores. Tal fue su éxito, que hasta el día de hoy seguimos usando UML como el lenguaje, para representar el diseño de artefactos de nuestras aplicaciones.

A la hora de implementar este lenguaje, ¿qué tan rigurosos debemos ser?

El propósito de UML es ser una herramienta y como tal, se debe tener en cuenta que su estricta utilización, irá directamente relacionada con el contexto en el que se esté trabajando, por ejemplo, si estamos trabajando en una herramienta, que a partir de los modelos que se creen con UML, se genere código de forma automatizada. Es en este caso, donde usaremos UML de forma rigurosa, para que el código generado sea el que se necesita.

En el caso que se desee utilizar UML en el contexto de una reunión en la que se desee entender un problema y donde los modelos diseñados no sean documentados formalmente (por ejemplo en una pizarra en una reunión con el cliente), podemos entonces usar UML con mayor libertad, sin dar tanta importancia a los detalles, sino que basta con que expliquen el problema de una forma global. En este tipo de casos debemos usar lo mínimo que nos aporte una mejor interpretación del problema que se desea resolver. Existen ocasiones en que la notación oficial puede llegar a estorbar, en estos casos no hay que dudar en adaptar el lenguaje a las necesidades personales. Quizás puede ser mal visto el que no estemos usando UML con todas sus sutilezas; pero en este caso ganamos flexibilidad y mientras la comunicación no se vea afectada, no existe mayor problema en alterar un poco este lenguaje. Hay que tener cuidado al momento de adaptarlo, no nos podemos exceder, ya que el diferir mucho la estructura oficial, puede resultar en que no se comprenda lo que se desee expresar.

Antes de usar UML, ¿qué nos debemos preguntar?

Ya se ha mencionado que UML lo usamos como una herramienta, y para que lo sea, tenemos que asegurarnos, que la utilización de este lenguaje **sea útil**. Después de todo, los diagramas que realizamos pueden ser visualmente atractivos; pero ningún cliente va a agradecer la belleza de los diseños que hemos generados con UML; lo que los clientes, usuarios, stakeholders, etc. quieren son aplicaciones que funcionan. Es por todo esto, que al momento de utilizar UML, debemos hacernos la siguiente pregunta: ¿Cuál será el beneficio de utilizarlo? y ¿Cómo su uso ayudará al momento de implementar el código?. Si estas preguntas no tienen una respuesta clara. Entonces, quizás el uso de modelos construidos con UML, no generan un real aporte, que justifiquen su utilización.

UML y los paradigmas de programación

El reconocido empresario de las tecnologías de la información `Tom Hadfield`, quien a la edad de doce años, creó junto a su padre `socckernet (1994)`, sitio que fue vendido a ESPN en 40 millones de dólares dijo:

> "Los lenguajes de objetos, permiten ventajas pero no las proporcionan. Para aprovechar estas ventajas hay que realizar el infame cambio de paradigma. (¡Sólo asegúrese de estar sentado al momento de hacerlo!)"

La mayoría de las personas que podían reconocer la programación en esos años e incluso ahora, la asocia al paradigma de la programación imperativa. Antes de que el mundo comenzara a utilizar la POO de forma natural, se utilizaba la programación procedural, y esto era lo usual, lo que sabíamos. por esta razón un cambio en el paradigma suponía un esfuerzo que queda reflejado en la frase de Hadfield. Entonces estamos ante una herramienta que nos permite modelar, y por ejemplo, por medio de un `framework` llevaríamos rápidamente a código los diagramas que hemos diseñado, si están correctamente diseñados por supuesto. Todo esto gracias a que UML fue concebido para la POO. Entonces, ¿en qué situación nos encontramos ahora?. Se podría afirmar que un "nuevo" paradigma emerge, el paradigma de la programación declarativa, todo esto debido a que lenguajes como `javascript`, que toma fuerza en la cantidad de aplicaciones que existentes o que se estén desarrollando; pero hay que mencionar que este lenguaje, es un lenguaje funcional (hija del paradigma declarativo), y, además es un lenguaje imperativo. Entonces nos encontramos ante una mezcla de lo que es programación funcional y programación orientada a objetos. Quizás debamos comenzar a comprender que esta mezcla de paradigmas puede coexistir y debemos acostumbrarnos a ello, verlo como una nueva forma de construir, y, no verlo como un dolor de cabeza, hasta que nos acostumbremos a su uso. UML nos seguirá sirviendo para representar gráficamente problemas; pero quizás el enfoque cambie un poco, hacia la utilización de nuevas herramientas que mezcle lo funcional con lo orientado a objetos. Verán que en librerías como *React js* ya utiliza esta mezcla de paradigmas y que cada nueva versión de Java, incluye muchas herramientas que permiten usar la programación funcional. Como sea que termine esta história, UML aún nos entrega una interfaz entre lo que es abstracto y lo que podríamos llamar tangible.

En el siguiente ejemplo, se puede ver el uso de estructuras funcionales, dentro del lenguaje Java. En ambos casos el resultado será 3 y 4.

```
package cl.desafiolatam.ejemplos.padigmas;

import java.util.ArrayList;
import java.util.stream.Collectors;

public class Principal {

    public static void main(String[] args) {
        ArrayList<Integer> numeros = new ArrayList<>();
        numeros.add(1);
        numeros.add(2);
        numeros.add(3);
        numeros.add(4);

        System.out.println("Resultado imperativo");
        imperativo(numeros);

        System.out.println("Resultado funcional");
        funcional(numeros);

    }

    public static void imperativo(ArrayList<Integer> numeros) {
        for(Integer numero: numeros) {
            if(numero > 2) {
                System.out.println(numero);
            }
        }

    }

    public static void funcional(ArrayList<Integer> numeros) {
        numeros.stream().filter(x -> x > 2).collect(Collectors.toList()).forEach(System.out::println);
    }

}
```

El proceso de desarrollo de software

UML, es un lenguaje para modelar, por eso no asume la noción de un proceso; pero para poder realizar un buen desarrollo, hay que analizar el propio proceso de desarrollar aplicaciones.

Una visión del desarrollo en su nivel más alto sería como la que muestra la Imagen 3.

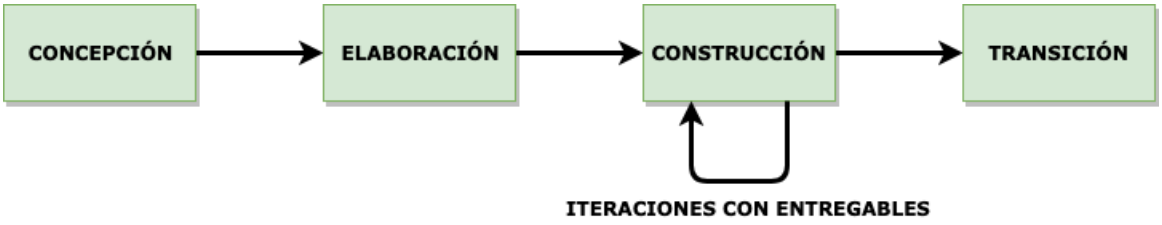

Imagen 3: Proceso de desarrollo de software.

Durante la **concepción**, se establece la razón de ser del proyecto, además del alcance del mismo. Es en esta etapa en donde se define si el proyecto se realiza o no.

En la **elaboración**, se detallan más profundamente los requerimientos, para así llegar a confeccionar una arquitectura base. Los diseños utilizados han de ser de alto nivel, es decir, sin gran nivel de detalle, destacando solamente lo principal. Dependiendo del nivel de `agilidad` del proyecto, esta etapa puede ser desde una reunión de un par de horas, hasta una entrega de documentos formales, con iteraciones. Al finalizar esta etapa, las preguntas: ¿Qué es lo que va a construir en realidad?, ¿Cómo lo va a construir?, ¿Qué tecnología empleará?, deben ser contestadas.

En la **construcción**, como se indica en la Imagen 3, la iteración es clave. No quiere decir que el las otras etapas no se pueda iterar; pero en esta etapa, cada iteración es un mini proyecto. Terminando con una demostración funcional al stakeholder. Con ello reducimos el riesgo, ya que si algo no ha salido como se quería, solamente está en juego la iteración y no el proyecto completo. La funcionalidad será incremental y la construcción de código será iterativa. **¿Cuándo se debe usar el desarrollo iterativo?, El desarrollo iterativo únicamente se debe utilizar en aquellos proyectos que se quiere que tengan éxito.**

Todas las técnicas de UML son útiles durante esta etapa; pero hay que ser cuidadosos, de no excedernos, como dice la siguiente cita:

> Los memorandos cuidadosamente escritos y seleccionados, pueden sustituir con toda facilidad a la tradicional documentación detallada del diseño. Esta última es sobresaliente en pocas ocasiones, excepto en algunos puntos aislados. Destaque estos puntos... y olvídese del resto.. "Ward Cunningham (1996)"

Durante la **transición**, no se hacen desarrollos para añadir funciones nuevas (a menos que sean pequeñas y absolutamente indispensables). Ciertamente, sí hay desarrollo para depuración. Un buen ejemplo de una fase de transición es el tiempo entre la liberación beta y]a liberación definitiva del producto.

La práctica de la ingeniería de software

Al describir el proceso de desarrollo de software, logramos poder ver las actividades estructurales generales, que son el esqueleto de la arquitectura para el trabajo de ingeniería de software. Con la finalidad de poder tener una mayor comprensión de esta práctica citemos un libro clásico *"How to solve it"*, escrito antes que existieran las computadoras modernas. Acá *George Polya*, escribió, sin saberlo, la esencia de la práctica de la ingeniería de software:

- Entender el problema (Comunicación y análisis).
- **Planear la solución (Modelado y diseño del software).**
- Ejecutar el plan (generación del código).
- Examinar la exactitud del resultado (probar y asegurar la calidad).

Centremos la atención en la planificación de la solución. Luego de entender el problema (o al menos creer que lo entendemos) y no puede contener las ganas de comenzar a desarrollar la solución; debemos realizarnos las siguientes preguntas:

- ¿Ha visto antes, problemas similares?
- ¿Hay patrones reconocibles en una solución potencial?
- ¿Hay algún software existente que implemente los datos, funciones y características que se requieren?
- ¿Ha resuelto un problema similar? Si es así, ¿Son reutilizables los elementos de la solución?
- ¿El problema, se puede dividir en problemas más pequeños? Si es así ¿Hay soluciones evidentes para estos?

Por eso, antes de comenzar a implementar la solución o generar modelos usando UML, nos ahorraremos una buena cantidad de tiempo, si tomamos un poco de aire y nos detenemos a respondernos las preguntas ya expuestas.

Un poco acerca de requerimientos

Los requerimientos son una especificación de lo que debe ser implementado. Estos son descripciones de cómo el sistema se debe comportar, de las propiedades y atributos del mismo. Deben ser una restricción del proceso de desarrollo del sistema. Existen dos clases de requerimientos, los requerimientos funcionales y los no funcionales.

Requerimientos funcionales (qué hacer)

Son las descripciones explícitas del comportamiento que debe tener el sistema y qué información debe manejar. Nos muestran las capacidades o cualidades que debe tener el sistema. Se enfoca en cuál debe ser el comportamiento de la solución y qué información debe manejar. Ejemplos de este tipo de requerimientos pueden ser los siguientes:

- El sistema deberá ser capaz de generar el reporte anual.
- Debe ser capaz de ejecutar procesos nocturnos de reportería.
- Mantendrá los datos de ventas actualizados.
- Debe disponer de la información detallada de los proveedores.
- Entre otros.

De este tipo de requerimientos, se extraen los casos de uso.

Requerimientos no funcionales (cómo hacerlo)

Describen eficiencia, seguridad lógica y de datos, usabilidad o cualquier tipo de requerimiento que no se refiera a las funcionalidades del qué se debe hacer. Son aquellos que nos define el como debe efectuar la tarea, en este tipo de descripción entran detalles como el rendimiento de los recursos:

- Los permisos de acceso al sistema podrán ser cambiados solamente por el administrador.
- El sistema debe ser capaz de operar adecuadamente, con hasta 100.000.000 usuarios con sesiones concurrentes.
- El tiempo de aprendizaje del sistema por usuario; deberá ser menor a 4 horas.
- Entre otros.

Tipos de diagramas en UML

No existe una línea que divida claramente UML; pero podemos reconocer los diferentes tipos de diagramas, para poder apreciar el aspecto del sistema que se intenta representar.

Tipos Estructurales:

- **Diagrama de clases.**
- Diagrama de paquetes.
- Diagrama de objetos.
- Diagrama de componentes.
- Diagrama de estructura compuesta.
- Diagrama de despliegue.

Tipos comportamiento:

- Diagrama de actividad.
- **Diagrama de secuencia.**
- **Diagrama de casos de uso.**
- Diagrama de estado.
- Diagrama de comunicación.
- Diagrama de interacción.
- Diagrama de sincronización.

Los diagramas que están destacados, se detallarán en los capítulos posteriores. Esto no se debe a que los demás no sirvan o sean menos importantes, más bien, se trata de poder utilizar lo mejor de UML, para resolver la mayoría de los problemas de diseño de software.

> De verdad, sólo requieres un 20% de UML, para modelar el 80%, del diseño que necesitarás en un proyecto – ágil o no". - Grady Booch (2018).

Los casos de uso

Competencias

- Entender los tipos de requerimientos.
- Comprender el concepto de casos de uso.
- Construir caso de uso.
- Identificar los casos de uso, para una problemática y representarla con la notación UML.

Introducción

Durante mucho tiempo, cuando era utilizado el paradigma procedural de forma popular, incluso cuando la POO comenzaba a ser muy conocida y utilizada. Los escenarios eran representados de forma muy rústica. las personas se auxiliaban de escenarios típicos que les ayudaban a comprender los requerimientos. Muchas veces se construían; pero no se documentaban, Ivar Jacobson, uno de los tres creadores de UML, elevó la visibilidad del caso de uso (su nombre para un escenario) a tal punto, que lo convirtió en un elemento primario de la planificación y el desarrollo de proyectos de software.

Uso de la herramienta StarUML

Esta herramienta nos permite realizar todos los tipos de diagramas que se explicarán en esta unidad, su uso es libre aunque si lo pagamos tendremos acceso a más opciones, para los que deseen ahondar en el diseño con esta herramienta existen planes mensuales y anuales. Para efectos de uso en la unidad, la versión libre nos será suficiente.

Para instalar la herramienta, debemos ir a la siguiente URL: http://staruml.io/download

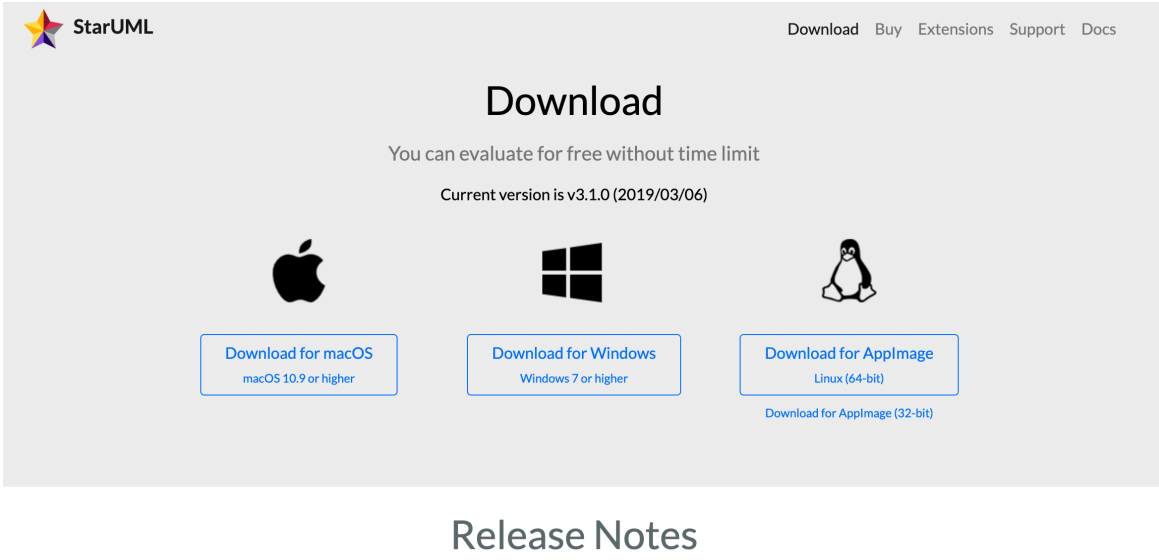

Imagen 4: Instalación herramienta StarUML.

Descargamos la versión de nuestro sistema operativo e instalamos como cualquier otro programa.

Una vez instalado podemos entonces comenzar a utilizarlo.

Diagramas de casos de uso (Cuentan una historia)

En el año 1994, Jacobson no solo logró introducir los casos de uso como un concepto para ayudar en los primeros pasos del desarrollo de aplicaciones, además, desarrollo una representación gráfica de este hecho y lo llamó El **Diagrama de Casos de Uso**, y este diagrama ahora forma parte de UML.

En la herramienta, nos vamos a model -> add diagram -> Use Case Diagram.

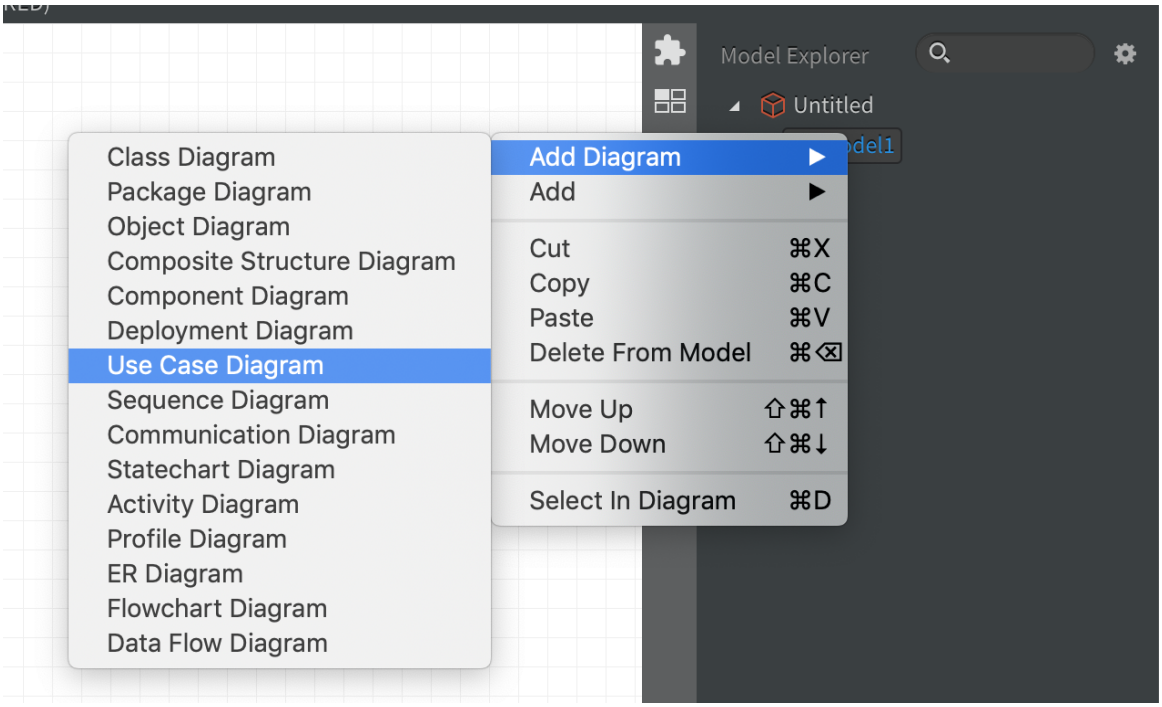

Imagen 5: Agregar diagrama de Casos de Uso.

Una representación de un diagrama de casos de uso sería el siguiente:

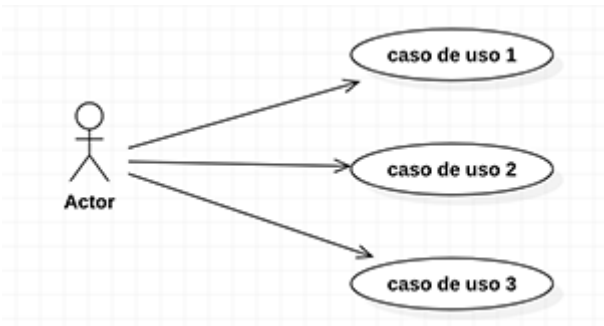

Imagen 6: Diagrama de Casos de uso básico.

Casos de uso (CU)

Un caso de uso es, en esencia, una **interacción típica entre un usuario y un sistema de software**. considere la aplicación con la que se escriben estas líneas que usted lee. Dos casos de uso típicos serían "poner una parte del texto en negritas" y "Borrar el texto seleccionado". Por medio de estos ejemplos, se puede uno dar una idea de ciertas propiedades de los casos de uso.

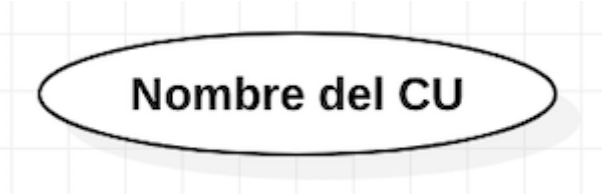

Imagen 7: Notación de un caso de uso.

- Capta alguna función visible para el usuario.
- Puede ser pequeño o grande.
- Logra un objetivo discreto para el usuario.

Usualmente, el caso de uso se extrae de las interacciones, que los potenciales usuarios del sistema tengan con la aplicación que se desee construir. Cada una de estas, se debe abordar de forma discreta, darle un nombre y escribir una breve descripción. No hay que detallar tan profundamente esta interacción, todo esto dependiendo de la cantidad de ramificaciones, de las que esté compuesto el caso de uso, se podrá más adelante, obtener mayores detalles que pueden resultar en nuevos casos de uso.

- **Objetivos del usuario vs interacciones con el sistema.**
  Cuando se están tomando los requerimientos, a veces es fácil confundirse entre los objetivos que el usuario tiene y las **interacciones que hará con el sistema**. Supongamos el desarrollo de un sistema de ventas. Surgirán interacciones como: "Ingresar un producto", "actualizar precio". etc. Esto difiere de los objetivos que pueda tener el usuario como, "mantener la información de los precios actualizada", "Garantizar que esté disponible la información de los productos". En ambos casos tenemos requerimientos del usuario, pero la granularidad es diferente, ya que las interacciones son más sencillas de implementar que los objetivos. Cuando estamos reuniendo los casos de uso del sistema, debemos mantener una granularidad más o menos similar, es decir, el nivel de detalle y complejidad de cada caso de uso, debe ser similar. Finalmente como lo definimos, mantendremos el foco de la captura de los casos de uso, centradas en las interacciones, sin perder de vista los objetivos, en el caso que sólo tengamos los objetivos del usuario, debemos granular esta información, en interacciones necesarias para que se cumplan los objetivos.
  Como recomendación, es bueno tener varios casos de uso por objetivo, al menos los objetivos principales, ya que en las iteraciones que vayamos haciendo, se recibe mucho mejor que el usuario vea como se van cumpliendo sus objetivos (aunque lo más probable es que estos objetivos vayan evolucionando a medida que el sistema avanza).

Actores

Empleamos el término actor para llamar así al usuario, cuando desempeña ese papel con respecto al sistema. Los actores llevan a cabo casos de uso. Un mismo actor puede realizar muchos casos de uso; a la inversa, un caso de uso puede ser realizado por varios actores. **No es necesario que los actores sean seres humanos**, a pesar de que los actores estén representados por figuras humanas. El actor puede ser también un sistema externo que necesite cierta información del sistema actual. Todos los casos de uso tratan sobre funcionalidad requerida externamente. Si el sistema de contabilidad necesita un archivo, entonces ése es un requerimiento que debe satisfacerse, en donde el actor sería el sistema de contabilidad.

Imagen 8: Notación de un actor.

Relaciones

UML, define relaciones de estereotipos y generalización. Con estas relaciones, podemos ver gráficamente el como interactúan los casos de uso y los actores, para una mejor comprensión del escenario. A continuación definiremos cada una de ellas.

- **Estereotipo:** `<<comunicate>>`

  Esta relación es la que más veremos en los CU, como estereotipo se representa por `<<comunicate>>`; pero generalmente este estereotipo no va escrito. Es una relación de asociación que nos muestra la interacción entre un actor y el caso de uso.

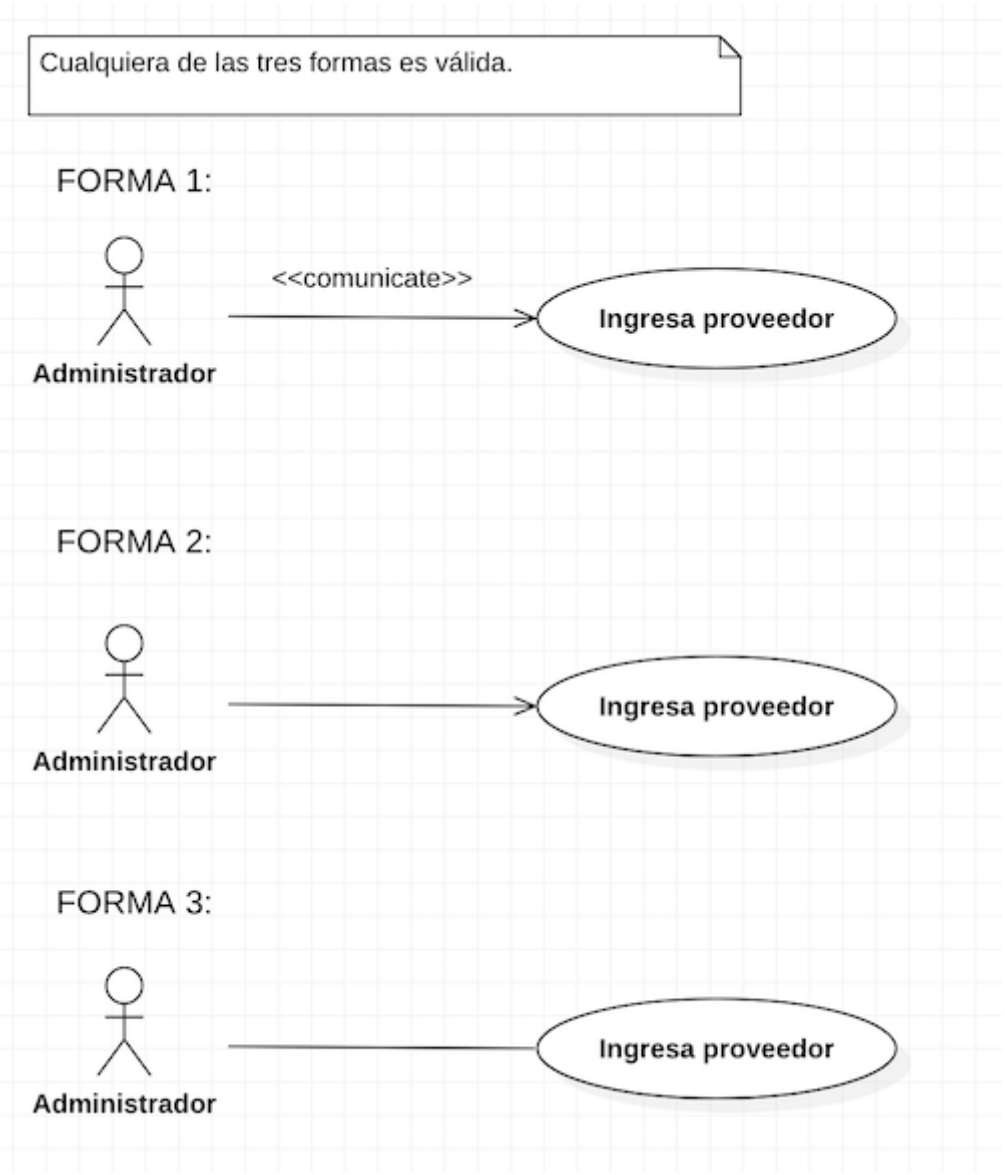

Imagen 9: Relación de comunicación de asociación.

- **Estereotipo:** `<<include>>`
  En términos muy simples, cuando relacionamos dos casos de uso con un “include”, estamos diciendo que el primero (el caso de uso base) incluye al segundo (el caso de uso incluido). Es decir, el segundo es parte esencial del primero. Sin el segundo, el primero no podría funcionar bien; pues no podría cumplir su objetivo.

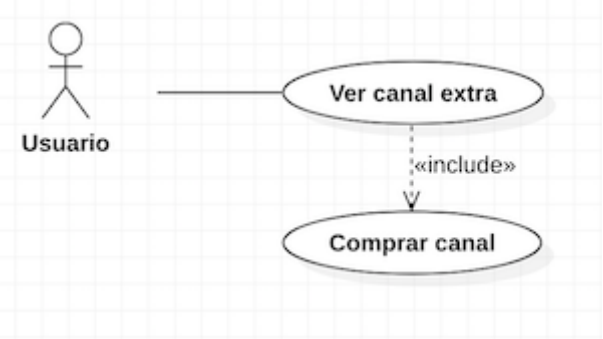

Imagen 10: Relación de inclusión.

- **Estereotipo:** `<<extend>>`

  Un caso de uso puede tener una extensión que no sea indispensable, pero sería bueno para la comprensión de lo que se desea desarrollar, que esta extensión sea expresada en el diagrama, es en este caso (no abusar), en donde haremos esta relación. Es decir, **el caso de uso base, puede funcionar perfectamente, si no se realiza el caso de uso del cual extiende.**

  Notamos en la Imagen 11, que la dirección de la flecha es en otro sentido, esto se debe a que así es la notación de la relación de extensión, el sentido va desde el caso de uso que se extiende, hasta el caso de uso que la consume.

  Debemos tener muy claro que esta extensión, no tiene que ver con la técnica de herencia en el paradigma de orientación a objetos, la relación de extensión y la herencia, son dos cosas completamente distintas, ya que se encuentran en contextos diferentes a pesar de su similitud en sus denominaciones.

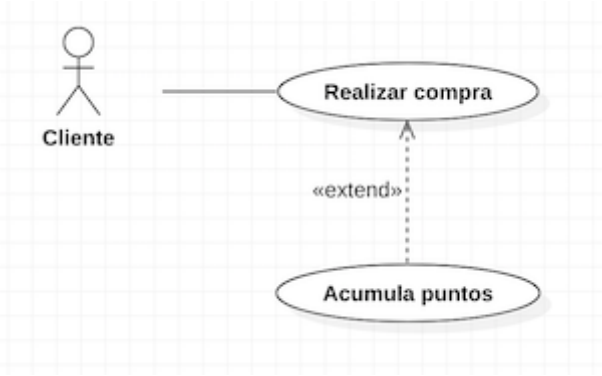

Imagen 11: Relación de extensión.

- **Generalización**

  Cuando hablamos de generalización, esta vez sí que nos estamos refiriendo, a algo muy parecido de lo que hace la herencia en la orientación a objetos, pero esta vez en el contexto de comprender el escenario. Cuando creamos a un actor y un caso de uso, si este caso de uso es una abstracción de otros casos de uso o el actor también podría ser una abstracción de muchos otros actores, podemos aplicar la generalización (otra vez, sin abusar). Así, si el "padre" contiene muchas cosas que un "hijo" que hace lo mismo pero más especializada, esta relación nos sería útil.

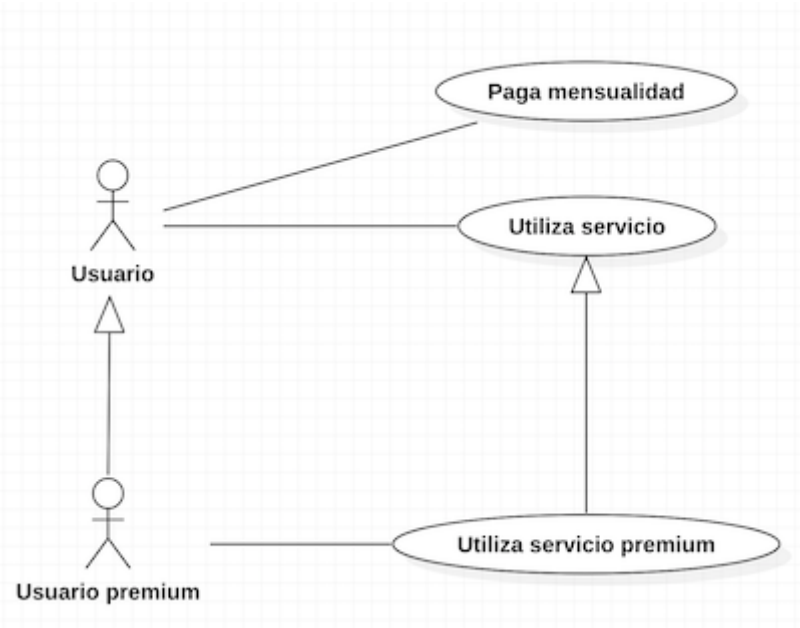

Imagen 12: Uso de la generalización.

- **Evitar abusos en la granularidad**
  Se debe tener presente que no hay que abusar del uso de estas relaciones, ya que se suele pensar que si el caso de uso se compone de varias piezas, estas deben ser expresadas en el diagrama. Hay que recordar que UML, es un lenguaje que lo utilizamos como herramienta, para lograr una mejor comprensión de lo que se debe desarrollar, complicarlo con complejos diagramas no es el objetivo. Es por eso que no haremos cosas como lo que se representa en la Imagen 13, y, nos enfocaremos siempre en lo suficientemente importante para la comprensión del escenario al que estamos representando, se podría tomar como referencia y sobre todo en un contexto ágil, que el diseño del escenario del caso de uso, cuando se implemente, no debería superar las tres semanas de desarrollo y que se trate solamente de aquellos detalles importantes para lograr que se cumplan estos plazos mentales.

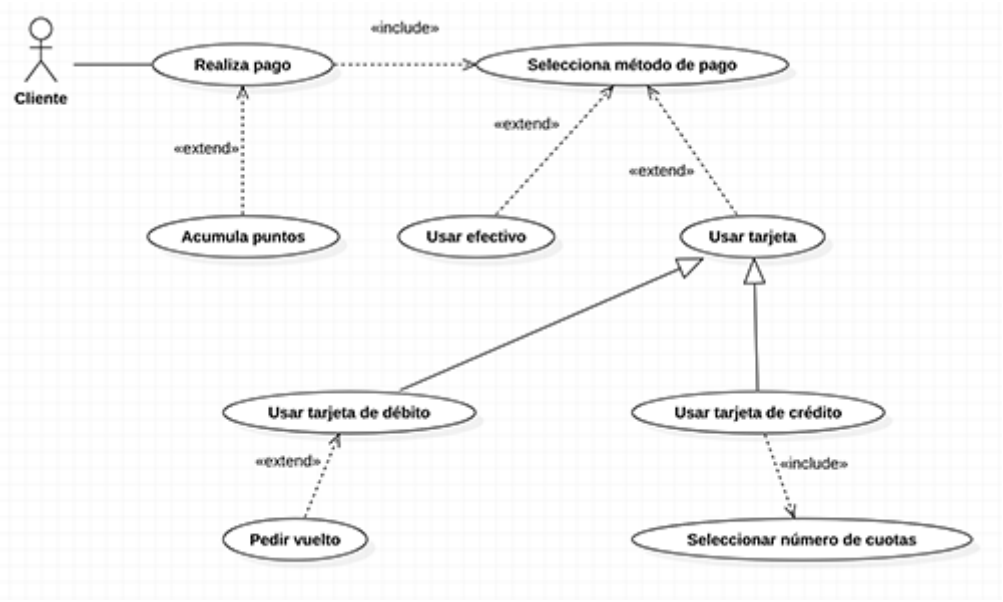

Imagen 13: Abuso de granularidad.

Ejemplos

A continuación, veremos direferentes ejemplos de diagramas de casos de uso, estos diagramas, pueden ser representados de diferentes maneras, siempre y cuando, se pueda comprender el escenario que se desea representar. Es por eso que las respuestas a estas preguntas no son únicas. Se recomienda diseñar los diagrámas de la forma más sencilla posible.

Ejemplo 1

Diseñar un diagrama de casos de uso, que exprese el escenario de una máquina expendedora de bebidas.

Solución:

Lo primero que podemos identificar son los actores que interactúan en este escenario:

- Usuario
- Proveedor

El usuario puede comprar el producto y el proveedor se encarga de retirar el dinero además de mantener la máquina con suficientes productos. Concluimos entonces que el escenario posee los siguientes Casos de Uso.

- Comprar producto.
- Recolectar dinero.
- Reponer productos.

Luego el Diagrama de Casos de Uso, sería el siguiente:

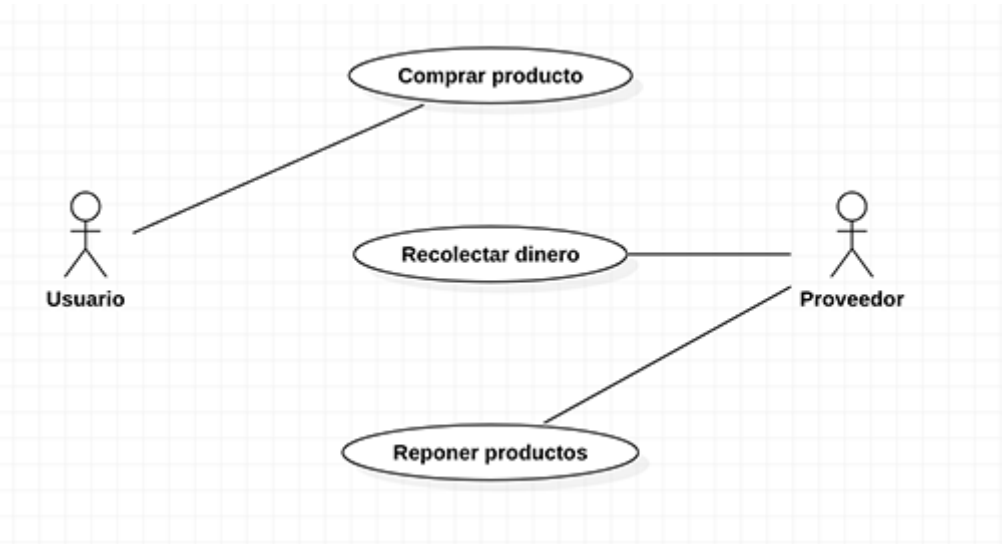

Imagen 14: Diagrama de Casos de Uso Ejemplo 1.

Ejemplo 2

Diseñar un diagrama de casos de usuo, que exprese el escenario que responde a los siguientes requerimientos:

- Un juego de teléfono móvil donde participan dos jugadores cada uno con su propia terminal.
- Cuando dos jugadores desean jugar, uno de ellos crea una nueva partida y el otro se conecta.
- El objetivo del juego es manejar una nave y disparar al contrario. Si uno de los dos jugadores acierta, la partida termina.
- Si uno de los dos jugadores deja la partida (o se pierde la conexión) la partida termina.

Solución:

Al igual que en el ejemplo anterior, debemos reconocer a los actores involucrados. Estos serían los siguientes:

- Terminal servidor, este se encarga de crear la partida.
- Terminal cliente, encargado de unirse a la partida que se ha creado

Podemos ver que ambos pueden dar término a la partida por tanto se concluyen los siguientes Casos de Uso.

- Crear partida.
- Unirse a la partida, necesita que la partida esté creada.
- Terminar partida, necesita que la partida esté creada.

Resultando en el siguiente Diagrama de Casos de Uso:

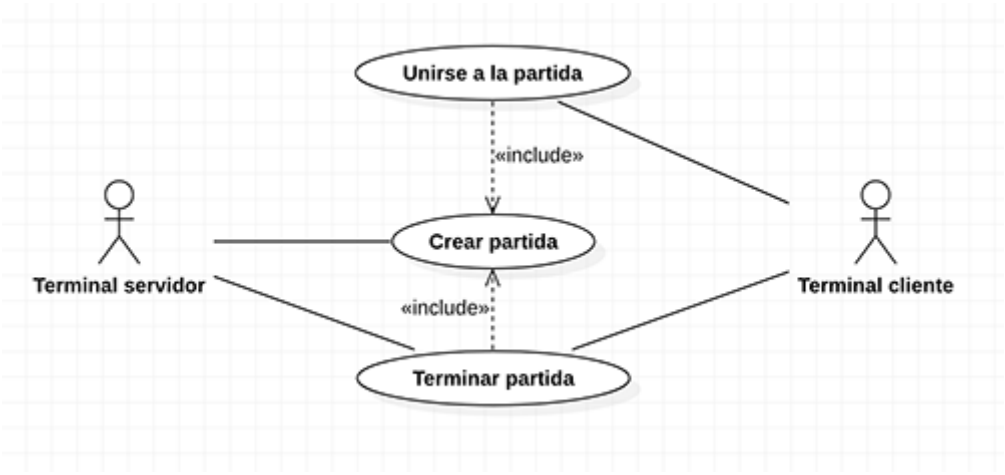

Imagen 15: Diagrama de Casos de Uso Ejemplo 2.

Ejemplo 3

Diseñar un diagrama de casos de uso, que exprese el escenario que responde a un sistema de ventas de entradas online.

Solución:

En este caso no disponemos de mucha información acerca de lo que se tiene que diseñar, excepto que es un sistema de ventas. Así que debemos usar la imaginación de un escenario típico.

Este tipo de sistema debería tener un usuario, este, tendría la opción de comprar una entrada. Quizás podría registrarse como socio. Como un actor no humano, tendríamos el sistema de pago, que usualmente debería ser un servicio aparte del sistema, ya que no es nuestra responsabilidad cumplir con el nivel que necesita una transacción monetaria. Finalmente necesitamos a alguien que revise las ventas realizadas.

De esto podemos obtener los siguientes Casos de Uso:

- Comprar entrada. (necesita cobrar).
- Cobrar.
- Registrarse como socio, opcional.
- Revisar ventas.

Como resultado de lo anterior, se obtiene el siguiente Diagrama de Casos de Uso:

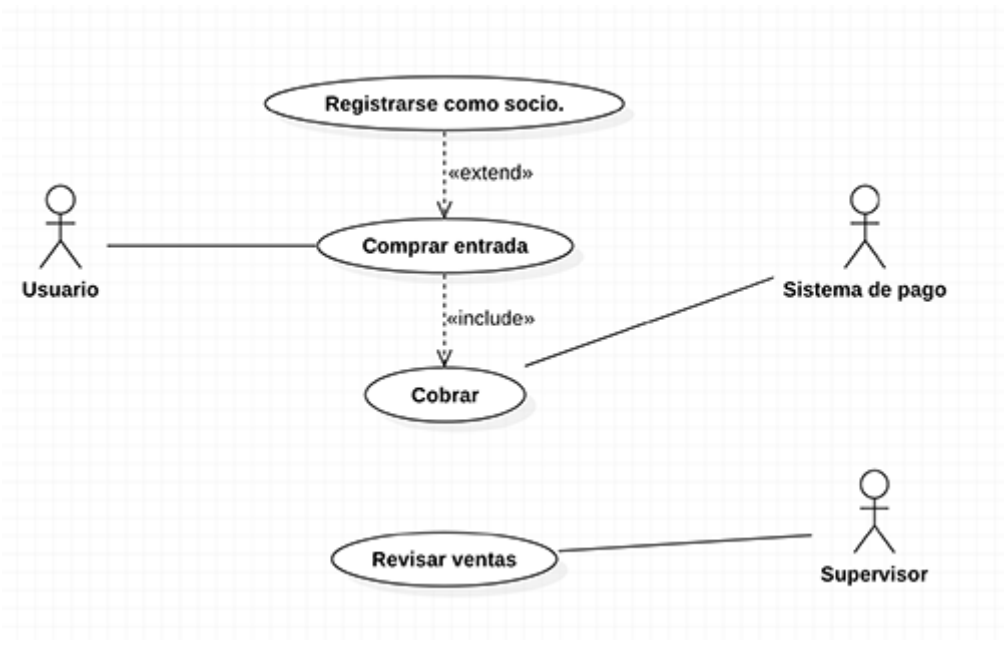

Imagen 16: Diagrama de Casos de Uso Ejemplo 3.

Diagramas de secuencia

Competencias

- Tener nociones de la vista de interacción.
- Conocer el concepto de los diagramas de secuencia.
- Conocer el papel de los roles y los mensajes.
- Construir diagrama de secuencia.
- Eficiencia en la construcción de los diagramas de secuencia.

Introducción

Extraer los casos de uso de una aplicación por medio de los requerimientos funcionales de un usuario, parece ahora una tarea simple. Pero ¿qué pasa con su comportamiento?. Tenemos las interacciones que asumimos que tienen los actores con el sistema; pero es posible analizar esas interacciones con un poco más de detalle para poder interpretar si el comportamiento que se pensó o se piensa, sea el correcto.

Diagrama de secuencia

En UML, existe la llamada "Vista de interacción", describe secuencias de intercambios de mensajes entre los `roles` que implementan el comportamiento del sistema. Esta visión proporciona una vista integral del comportamiento del sistema, es decir, muestra el flujo de control a través de muchos objetos, este tipo de vista se expresa en dos diagramas, el diagrama de colaboración y el diagrama que veremos a continuación: El diagrama de secuencia.

Un diagrama de secuencia, muestra un conjunto de mensajes, dispuestos en una secuencia temporal. Cada rol en la secuencia se muestra como una línea de vida, una línea vertical. A diferencia de otros diagramas, el diagrama de de secuencia al mostrarnos interacciones entre los roles, está dentro de los diagramas dinámicos. **Puede usarse un diagrama de secuencia, para mostrar las interacciones en un caso de uso o en un escenario de un sistema de software.**

En la herramienta, nos vamos a model -> add diagram -> Sequence Diagram.

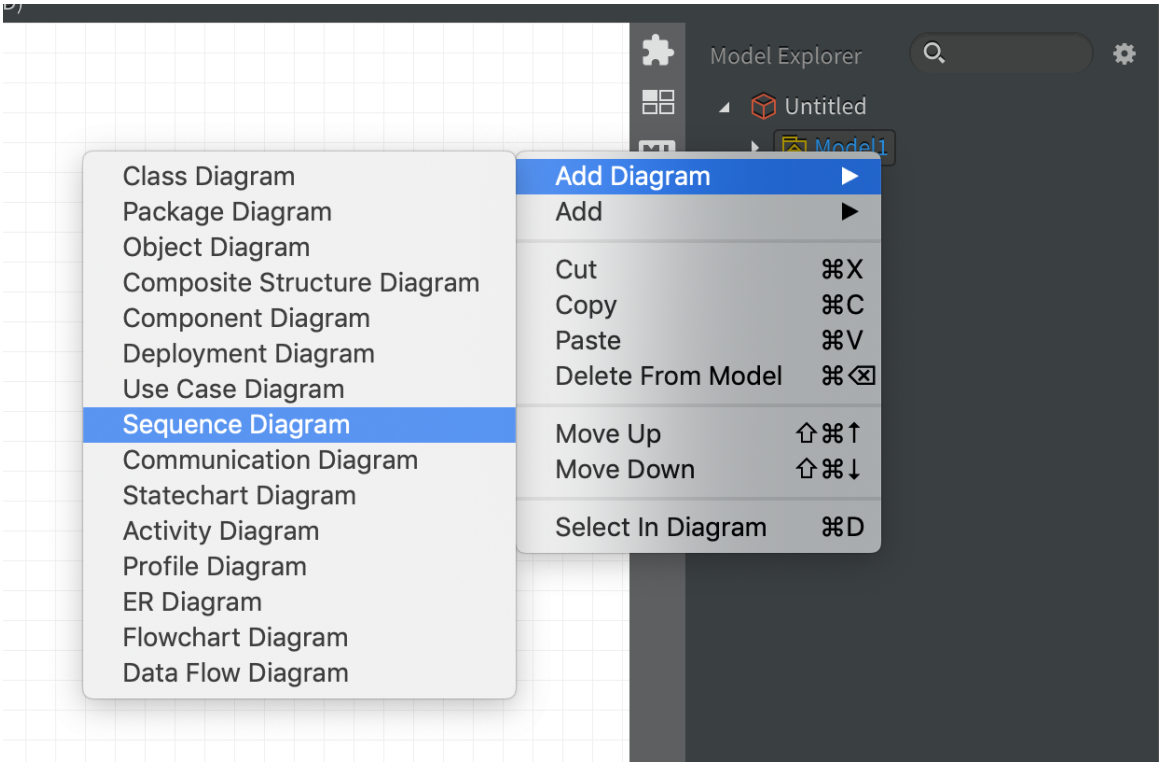

Imagen 17: Agregar Diagrama de Secuencia.

- **Rol**

  Es la descripción de un objeto, que desempeña un determinado papel dentro de una interacción.

- **Mensaje**

  Es la funcionalidad que permite la comunicación entre los roles. De acá, ya tendremos una idea de lo que serán los métodos y sus interacciones.

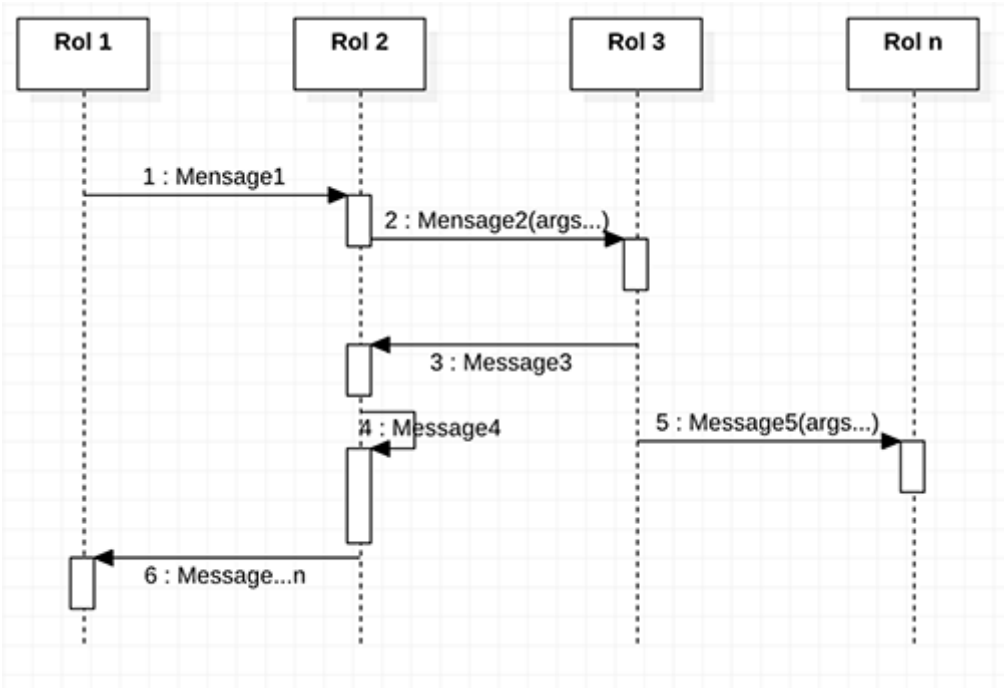

Imagen 18: Diagrama de secuencia.

Como vimos en un ejemplo anterior, en donde usamos los casos de uso para modelar el escenario de la compra de entradas online. Veamos ahora como poder llevar ese caso de uso a un diagrama de secuencia. De un diagrama de casos de uso, pueden salir muchos diagrámas de secuencia, por ejemplo, está la secuencia en el caso que, el usuario acepte o rechace la inscripción como socio, además de la interacción del administrador con el sistema. Es por eso, que se muestra un flujo normal y los demás se descartan. Hay que recordar que debemos usar solamente lo necesario para que la idea sea captada.

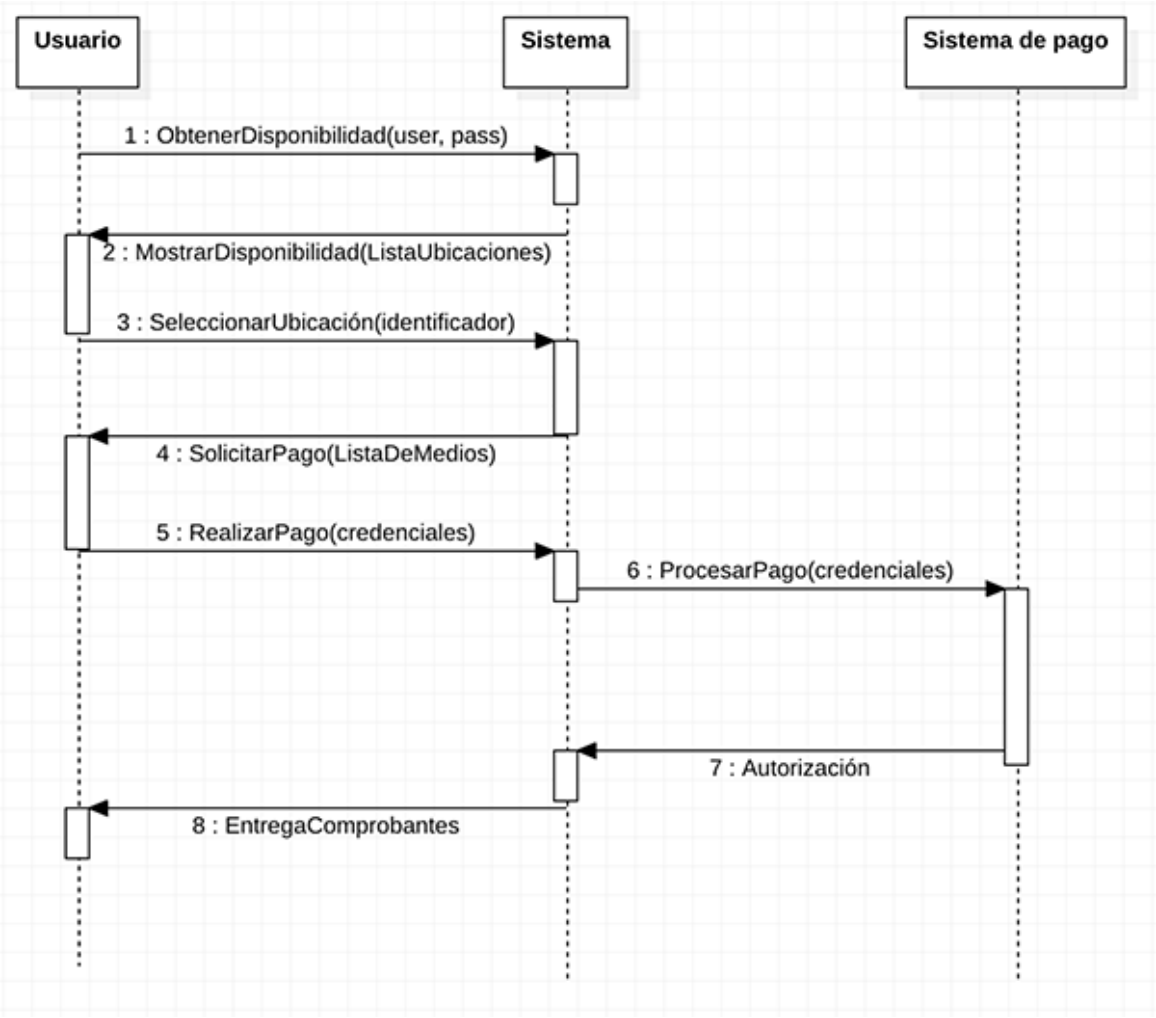

Imagen 19: Compra de entradas online, flujo normal.

Hagamos el flujo normal del administrador consultando las ventas.

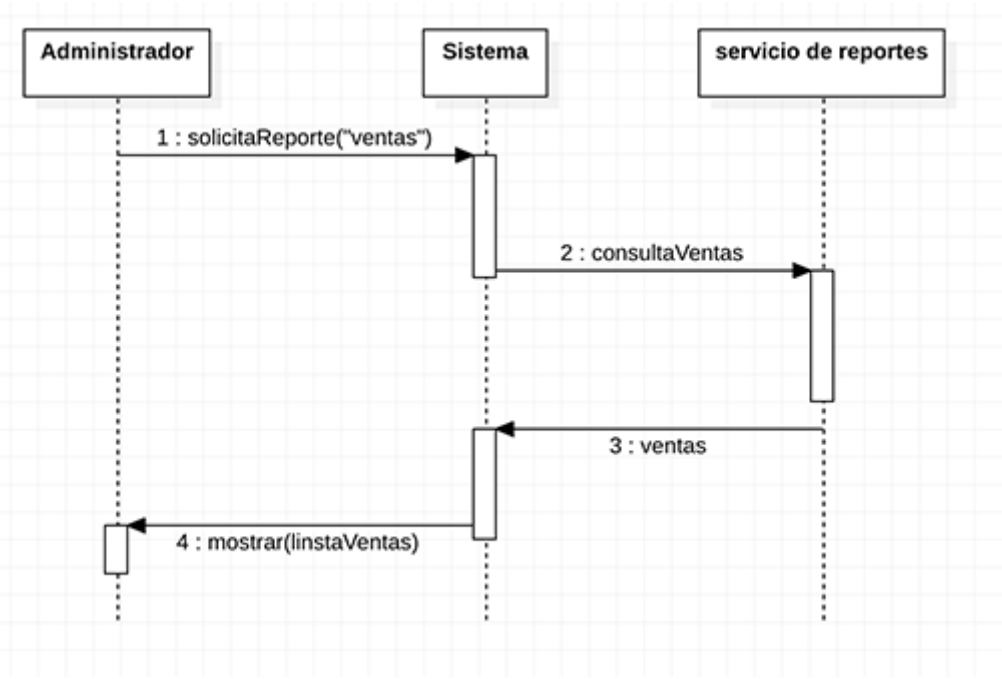

Imagen 20: Flujo normal, administrador consulta las ventas.

Se puede ver que algunos mensajes poseen argumentos y otros no. Notamos una similitud a la programación de métodos; pero hay que tener cuidado al pensar que deben estár la totalidad de los métodos que contendrán nuestras clases, ya que esto sería una complicación innecesaria en la etapa de toma de requerimientos. Lo que haya faltado por diseñar, se puede agregar en la siguiente iteración, a menos que se tenga el suficiente tiempo, además de la absoluta claridad en lo que se quiere desarrollar.

Podemos además como se ha mencionado, diseñar flujos alternativos. Un flujo alternativo, es aquél que no forma parte del problema principal, pero amerita que sea expuesto, por ejemplo, si es que pasa algún error en el flujo y saber como abordarlo.

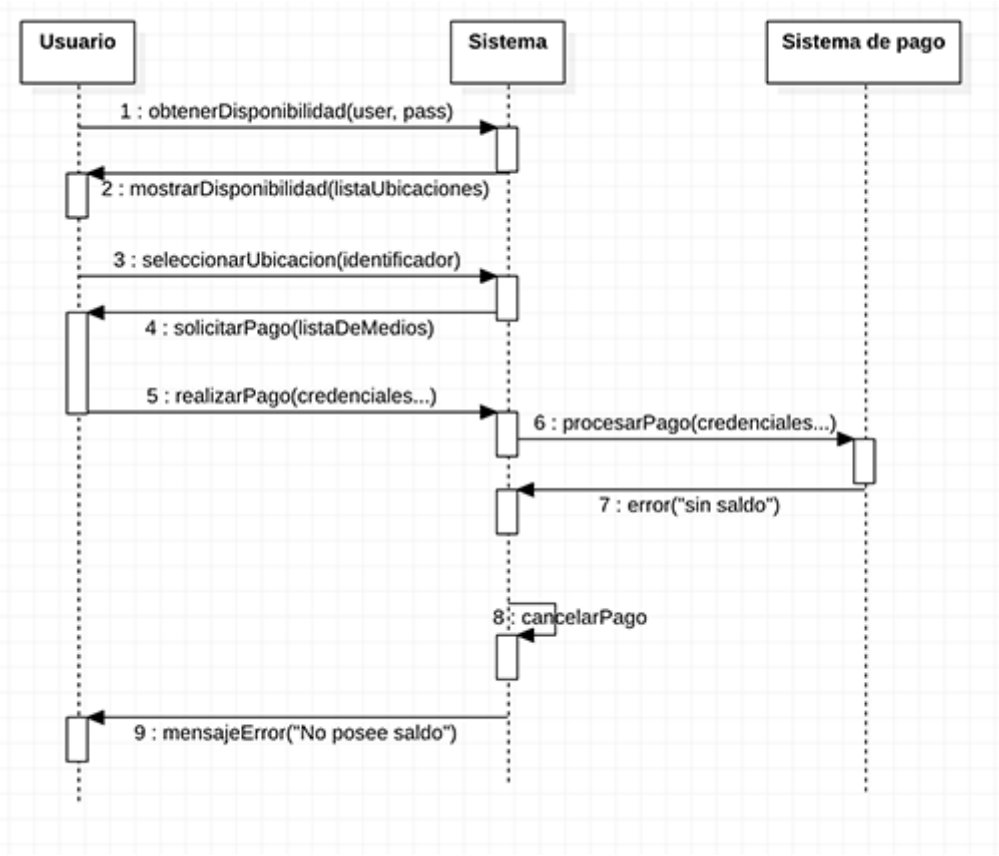

Imagen 21: Flujo alternativo, cuenta no posee saldo.

> "El código surge con naturalidad del diagrama de secuencia. En la práctica, generalmente me sirvo de un diagrama de secuencia para bosquejar la interacción y después hago algunos cambios a medida que lo codifico. Si la interacción es importante, entonces actualizo la gráfica de secuencia como parte de mi documentación. Si considero que tal gráfica no añadirá mucha claridad al código, archivo el borrador de la gráfica de secuencia en el archivero circular", - << Martin Fowler >>

Ya podemos entonces identificar las interacciones entre los roles. Tenemos a un usuario que requiere la compra de una entrada, entonces podemos imaginar que necesitamos un tipo de dato llamado entrada, quizás en este punto se nos ocurre la idea que puedan existir varios tipos de entradas y le preguntemos al especialista del negocio, aquellas dudas que surgen en este punto, antes que sea demasiado tarde y tengamos desarrollado algo erróneo.