2019

Herramientas de productividad HDP115

UNIDAD III: HERRAMIENTAS PARA ANÁLISIS Y DISEÑO

BLADIMIR DIAZ CAMPOS

UNIVERSIDAD DE EL SALVADOR | Facultad de Ingeniería y Arquitectura, Escuela de Sistemas Informáticos

Contenido

1 Notación UML 2.0

La versión 2.0 de UML actualiza la simbología de los diagramas de secuencia. Ya en UML 1.x se planteaba la representación de objetos de frontera y objetos de control. Sin embargo, en UML 2.0 se definen símbolos para denotar estas características.

Además, se definen símbolos derivados del aparecimiento de Patrones de diseño en la POO. Estos símbolos tienen significados diferentes según la tecnología, la arquitectura y los lineamientos técnicos definidos en el desarrollo de las aplicaciones.

Objetos de frontera

Los objetos de frontera representan los objetos que se comunican directamente con el actor. Usualmente se refieren a interfaces de usuario.

Objetos de control

Son los objetos que tienen como responsabilidad, controlar el flujo de los mensajes hacia los objetos internos del sistema.

Reciben las peticiones originadas en el objeto de frontera e invocan a los métodos de los objetos internos para dar una respuesta.

Objetos de entidad

Dependiendo de la tecnología, técnicas y patrones de diseño aplicados, los objetos de entidad se pueden referir a servicios de acceso a datos o a clases que representan tablas de la base de datos al interior de la aplicación (tal y como se hace con la técnica ORM).

2. El proceso

Como ya se mencionó, los diagramas de secuencia son una extensión de los casos de uso que incluyen detalles de implementación del escenario descrito en los casos de uso. Esto quiere decir que habrá un diagrama de secuencia por cada caso de uso descrito en el análisis.

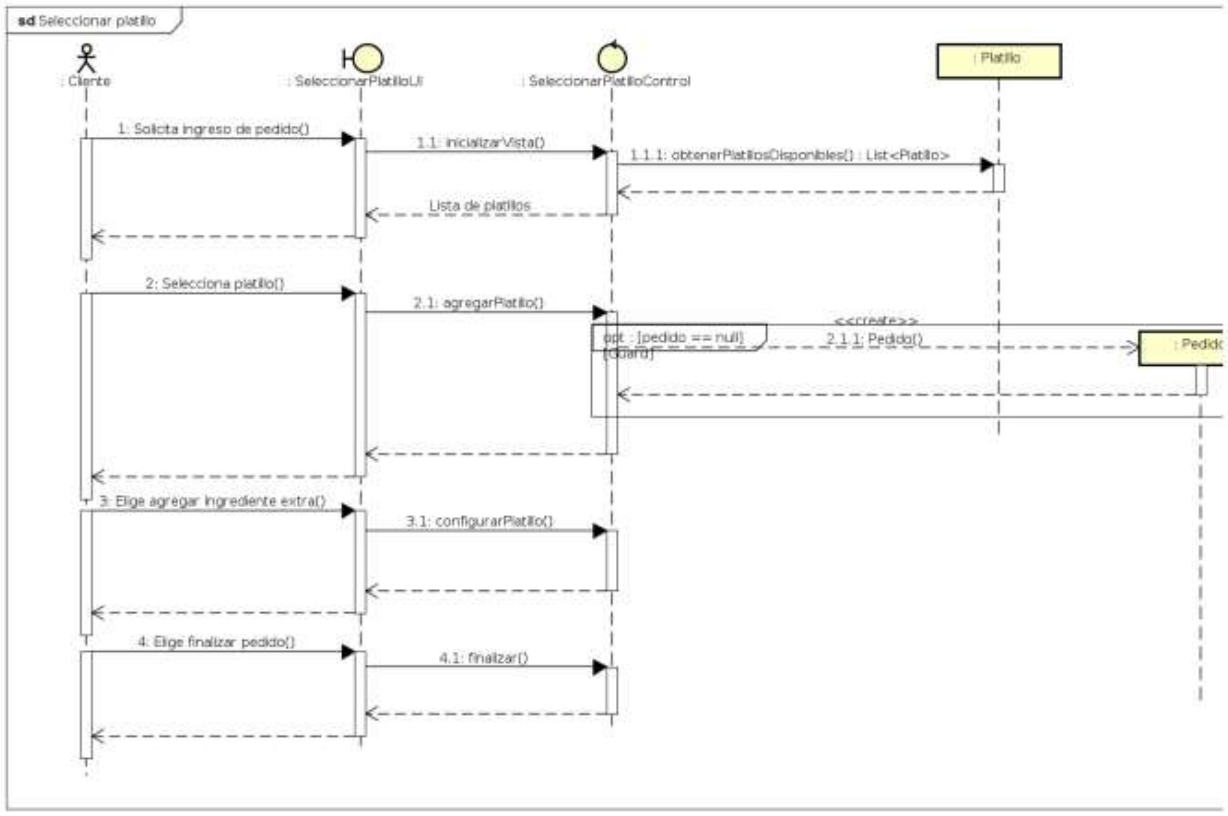
Normalmente el diccionario de caso de uso es utilizado como guion de la secuencia, sirviendo para identificar los actores que participan en la secuencia, los objetos y los mensajes.

En la mayoría de los casos, es necesario incorporar a los actores y objetos de las clases previamente identificadas, objetos de frontera y objetos de control.

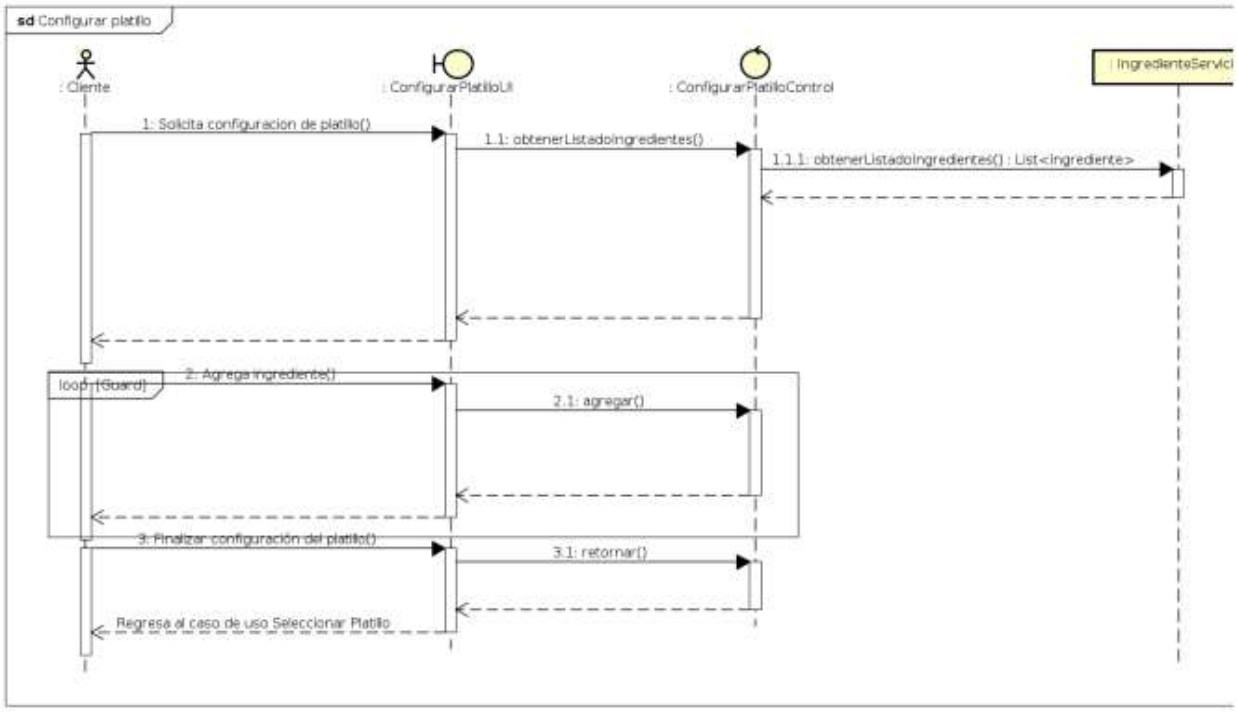
Los objetos de frontera representan las interfaces encargadas de interactuar con los actores. Los objetos de control, representan aquellos objetos que se encargan de invocar a los objetos del negocio, controlando el flujo de los mensajes.

Ejemplo

Retomaremos el ejemplo de Pedidos en línea a un restaurante y crearemos los diagramas de secuencia de los casos de uso Seleccionar platillo y Configurar platillo.



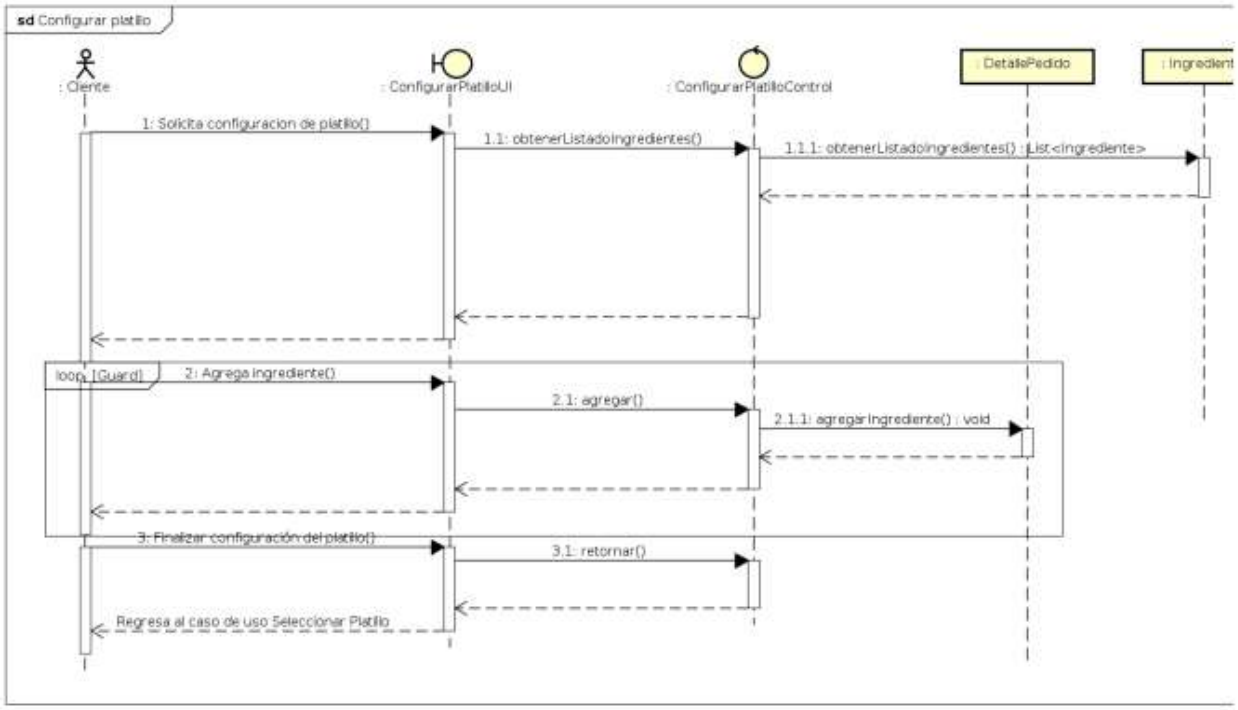
Observese que los flujos descritos en los diagramas corresponden a los pasos establecidos en el caso de uso correspondiente. Dado que el problema nos plantea que se trata de un sistema para realizar ordenes de comida en línea, no es posible suponer que los objetos de frontera jugarán un papel de control también.



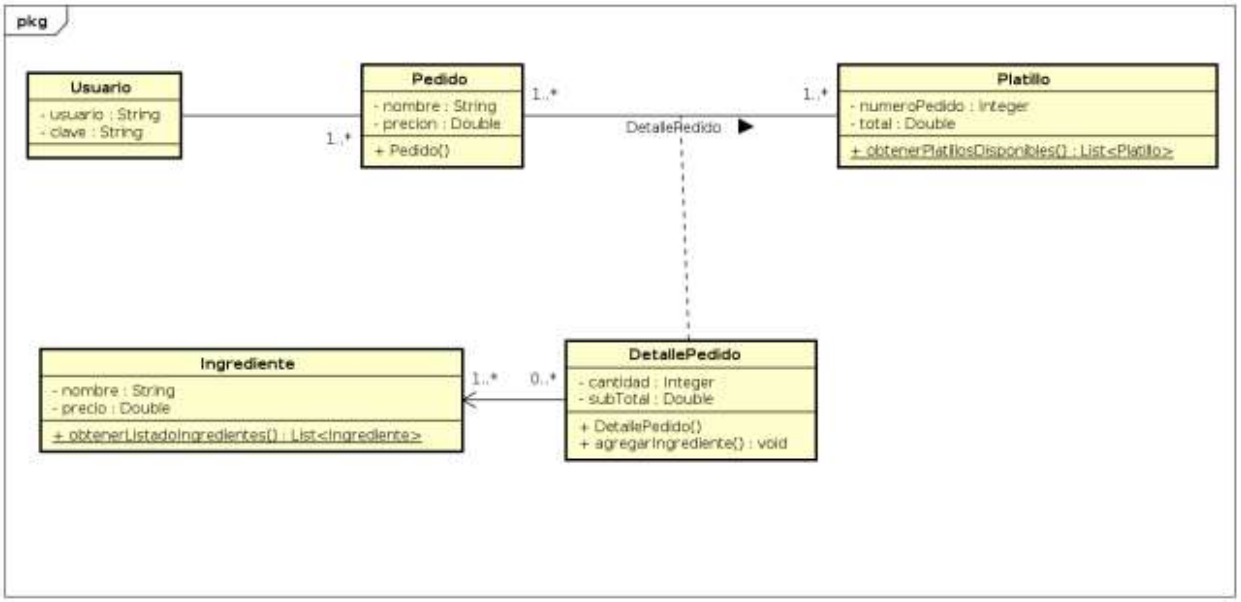
Nótese que, en el modelo de secuencias, la responsabilidad del acceso a los datos recae sobre las clases del modelo. Es decir, estas clases; además de definir los datos que manipulará el sistema, deberá contar con métodos a través de los cuales se acceda a la base de datos. Por ejemplo, el método guardar deberá tener una instrucción SQL de inserción de datos.

Por otro lado, el diseño asume que, cuando se agrega un nuevo ingrediente a un platillo seleccionado, será en la operación agregar de la clase de control que se agregará el objeto seleccionado previamente, a la lista de ingredientes del objeto de tipo *DetallePedido*. En esta operación también sumará al subtotal del detalle del pedido el costo correspondiente del ingrediente agregado.

Otra alternativa es crear una operación en la clase *DetallePedido* que realice ese proceso.



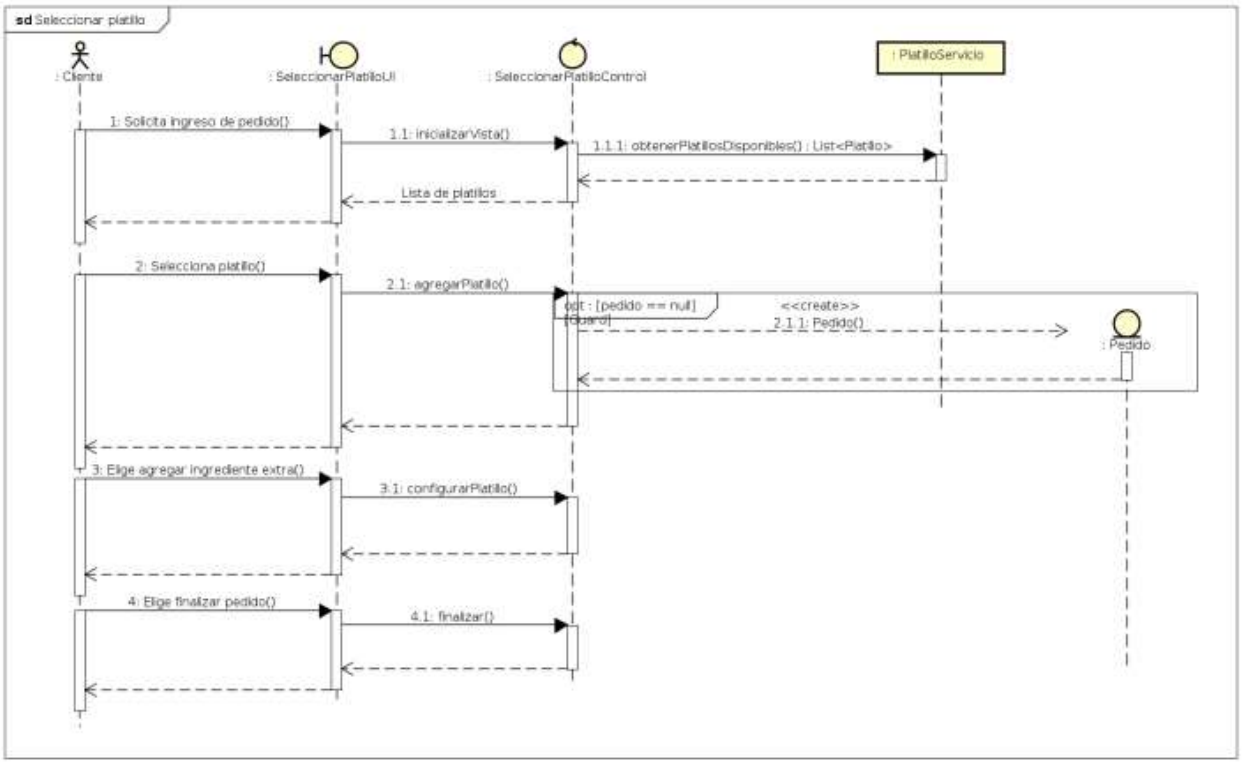
Después de identificar los métodos, el diagrama de clases resultante será:

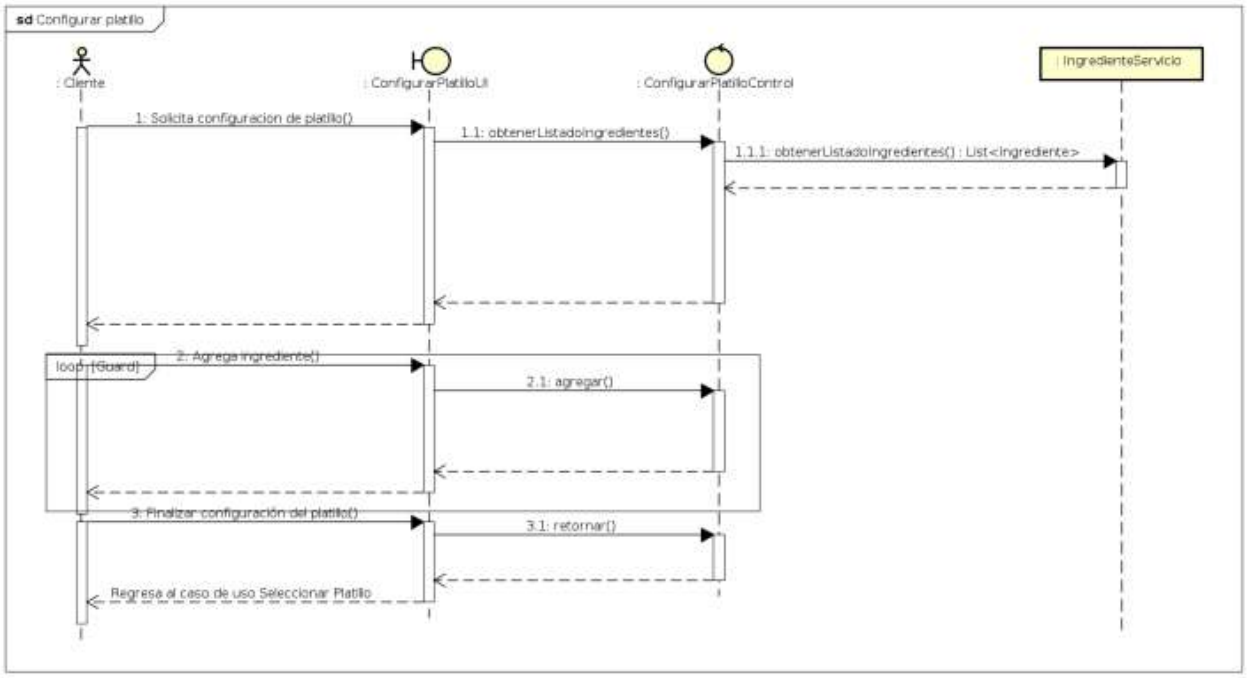


En este modelo, las clases del modelo son tomados como clases de entidad. En UML 2.0, esto es especificado agregando un estereotipo entity a las clases del modelo.

Otro enfoque es utilizar clases de servicio que se utilicen como expertos, como lo definen los patrones GRASP. Este enfoque surge de que, en determinados escenarios, el modelo de clases de entidad con operaciones resulta ineficiente en el uso de la memoria.

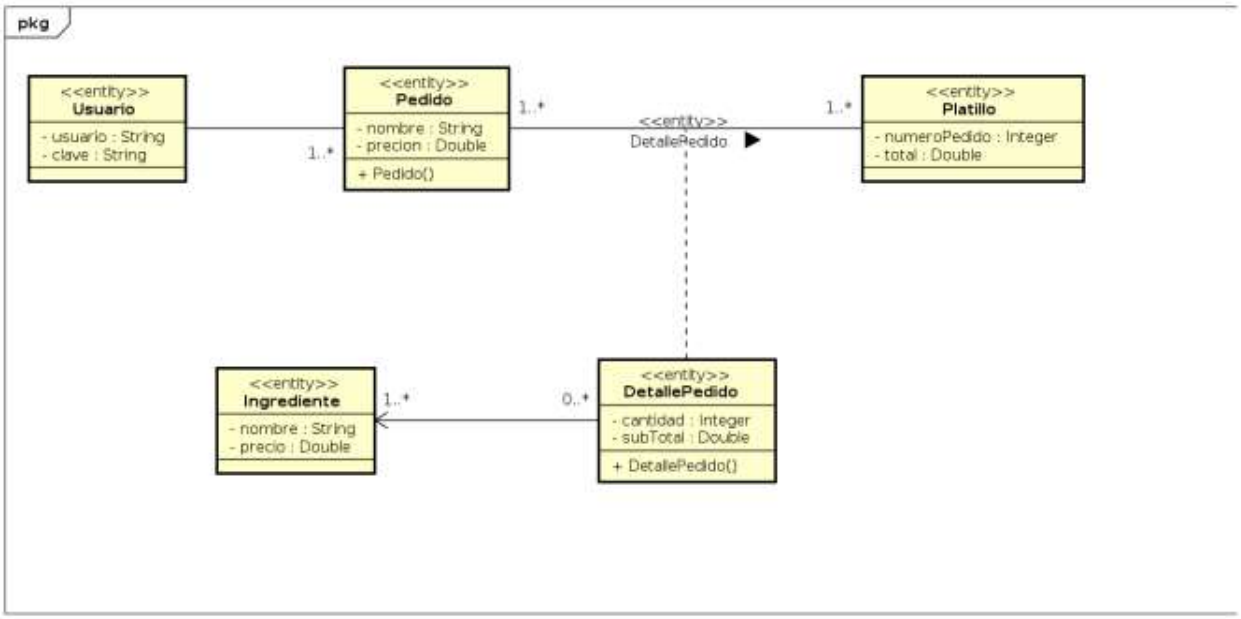
Bajo este enfoque, los diagramas de secuencia quedan como sigue.

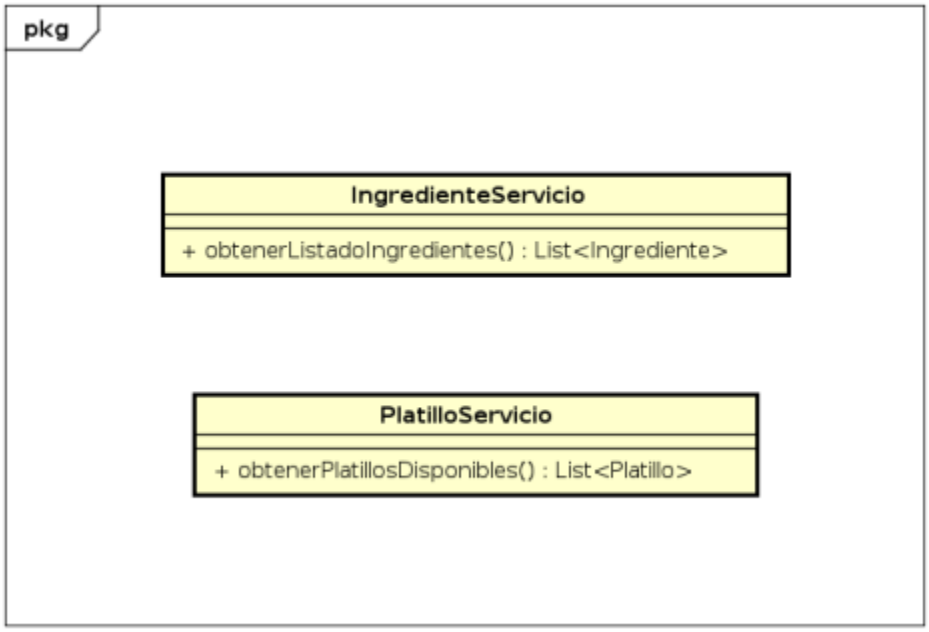




Nótese que en esta solución no es posible delegar la operación agregarIngrediente a la clase de entidad DetallePedido, puesto que las clases de entidad solamente definen las estructuras de datos a utilizar, y tienen como únicas operaciones sus constructores.

Los diagramas de clases de las clases, de las clases de entidad y de servicio quedarán como sigue.





El enfoque de diseño en el que las clases del negocio también incluyen operaciones donde se implementan las reglas del negocio y la persistencia derivó en lo que ahora conocemos como el patrón Active Record, que acude a algunas técnicas de programación para reducir el uso excesivo de memoria a la hora de manipular los objetos.

Una variante (alternativa al enfoque en el que se utilizan clases de servicio) es dejar las operaciones donde se implementan las reglas del negocio y la persistencia en las clases control.

La decisión entre dejar la lógica del negocio y la persistencia en las clases controladores dependerá de la arquitectura interna que se adopte, particularmente la implementación y uso de frameworks de persistencia y su integración con otros patrones de diseño. Esto deberá ser especificado como parte del diseño y los lineamientos de desarrollo.

Con el tiempo, se crearon una serie de definiciones que sirven para identificar estos componentes y organizarlos de manera ordenada:

1. Modelo del negocio o modelo de clases del negocio. Se le llama modelo del negocio al conjunto de clases que modelan los conceptos más importantes del problema. Su nombre proviene de la necesidad de diferenciar estas clases del resto de clases del sistema: Clases

de vista (por ejemplo: ventanas, botones, text box, etc.), clases controladoras, clases de aplicación, etc. Estas clases tienen dos responsabilidades fundamentales:

   a. Definir las estructuras de datos que servirán para soportar los datos más importantes de la aplicación. Es decir, tener una estructura en la cual podamos guardar temporalmente los datos para posteriormente ser almacenados y mostrar los datos cuando sean recuperados del depósito utilizado para el almacenamiento.
   b. Contener las operaciones que implementen las reglas del negocio y la persistencia de los datos. Estas operaciones posteriormente se delegaron a las llamadas clases de servicio.

2. Clases de entidad: Las clases de entidad son clases del negocio en los que solamente se implementan las estructuras de datos. Si se utiliza el patrón Active Record, estas clases también tendrán las operaciones de persistencia.
3. Persistencia de datos: Llamamos persistencia de datos a la captura, almacenamiento y recuperación de los datos en un medio permanente (bases de datos o archivos físicos).
4. Reglas del negocio: Son los procesos para transformar los datos. Por ejemplo: calcular un interés de una cuenta, aplicar un recargo por mora, permitir la inscripción de una asignatura únicamente si se ha ganado el prerrequisito, etc. Las reglas del negocio usualmente se encuentran en los diccionarios de casos de uso.

2. Patrones de diseño

En cualquier ámbito de la actividad humana, es posible encontrar patrones de comportamiento de los fenómenos. Un patrón de comportamiento se refiere a que en un fenómeno podemos identificar variables que tienen un comportamiento constante y predecible en el tiempo.

Si estos fenómenos representan problemas que debemos resolver, podemos establecer una forma de resolverlos, que en consecuencia también tendrá un patrón de comportamiento.

El concepto de patrón de diseño tiene su origen en la arquitectura. El arquitecto Christopher Alexander propuso el aprendizaje y uso de una serie de patrones para la construcción de edificaciones. La idea fundamental era que es posible diseñar soluciones a los problemas comunes y recurrentes en los proyectos de construcción, y de esa forma tener una solución, antes de que ocurran en un proyecto específico.

En sus propias palabras: "Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, para describir después el núcleo de la solución a ese problema, de tal manera que esa solución pueda ser usada más de un millón de veces sin hacerlo ni siquiera dos veces de la misma forma."

En la ingeniería de software, fue hasta principios de la década de los 1990 que se adoptó este concepto. La publicación del libro Design Patterns: Elements of Reusable Object-Oriented Software, fue el hito para introducir este concepto al proceso de desarrollo orientado a objetos.

En él se describen de manera formal una serie de patrones de diseño de software. Los patrones de diseño son un nivel más alto de abstracción del diseño del software. Persiguen establecer estrategias de solución a los problemas comunes a cualquier proyecto de desarrollo de software. De esta manera, sirven como una guía para el desarrollo del software, resolviendo el problema de la hoja en blanco e incorporando las mejores prácticas de diseño.

2.1 Categorías de patrones

1. Patrones de arquitectura. Son una guía para la solución de problemas más comunes del diseño arquitectónico del software. Especifican un conjunto de predefinido de componentes y sus responsabilidades, y una serie de recomendaciones para organizar dichos componentes. Los patrones que más notoriedad han ganado hoy en día de esta categoría son Model ViewController (MVC), Service Oriented Architecture (SOA), Active Record, Data Access Object (DAO), Data Transfer Object (DTO) y Service Locator.

2. Patrones de diseño: Definen micro-arquitecturas de componentes del sistema.
    a. Patrones creacionales. Son una guía para la solución de problemas de creación de objetos. Los patrones creacionales se basan en dos ideas fundamentales: Encapsular el conocimiento de qué clases concretas usa el sistema y esconder como son creadas y combinadas las instancias de esas clases. Este tipo de patrones son utilizados principalmente para resolver el problema de la persisntecia de los datos en aplicaciones. Web. Los patrones que más notoriedad han ganado en esta categoría son los patrones Dependency Injection y Singleton.
    b. Patrones estructurales. Son una guía para la solución de problemas derivados de diseños con altos niveles de abstracción. Particularmente, se ocupa de resolver problemas de composición y agregación entre clases y objetos. Los patrones que más notoriedad tienen en esta categoría son: Adapter (o Wrapper) y Bridge.
    c. Patrones de comportamiento. Son una guía para resolver el problema de interacción entre clases y objetos, y la definición clara de las responsabilidades que tendrá cada parte en dicha interacción. La solución a estos problemas, dan como resultado métodos de comunicación entre objetos y clases más flexibles. Uno de los patrones más importantes de esta categoría es el patrón Iterator
    d. Patrones de concurrencia. Son una guía para resolver el problema de la múltiple concurrencia. Nacen a partir de los problemas derivados en las primeras aplicaciones Web, donde se tienen muchos clientes solicitando realizar las mismas operaciones de forma independiente. El patrón patrones más importantes en esta categoría son: Active Object y Thread Pool.
3. Idiom (Dialectos). Describe detalles de la estructura y comportamiento de un componente.

Algunos de los factores que inciden en la elección de un patrón de diseño son: el tamaño de la aplicación, la cantidad de usuarios y los requerimientos no funcionales. Estos factores pueden

llevarnos incluso a combinar varios patrones. De hecho, es muy usual que el diseño arquitectónico de las aplicaciones Web esté basado en el patrón MVC, un patrón creacional (por ejemplo: Dependency Injection o Singleton) y otro patrón de arquitectura (por ejemplo: Service Locator, Active Record o Data Acccess Object).

2.2 Modelo vista controlador (MVC)

En esta asignatura nos centraremos en el estudio del patrón Model View Controller (Modelo Vista Controlador).

El patrón MVC nace como una solución a los problemas comunes a todos los proyectos de desarrollo: la escalabilidad (la capacidad para expandir la funcionalidad del software), y el alto costo de mantenimiento.

En el proceso de desarrollo tradicional se observó que la estructura del código era similar y presentaban los mismos inconvenientes en cuanto a escalabilidad y mantenimiento. Por ello, se propuso separar el código en tres capas, cada una con una responsabilidad específicas.

1. La capa Modelo. Tiene las siguientes responsabilidades:
    a. Define las estructuras que corresponden a los datos más importantes de la aplicación (Clases del negocio).
    b. Implementa las reglas del negocio (lógica del negocio)
    c. Implementa los servicios de acceso a la base de datos.

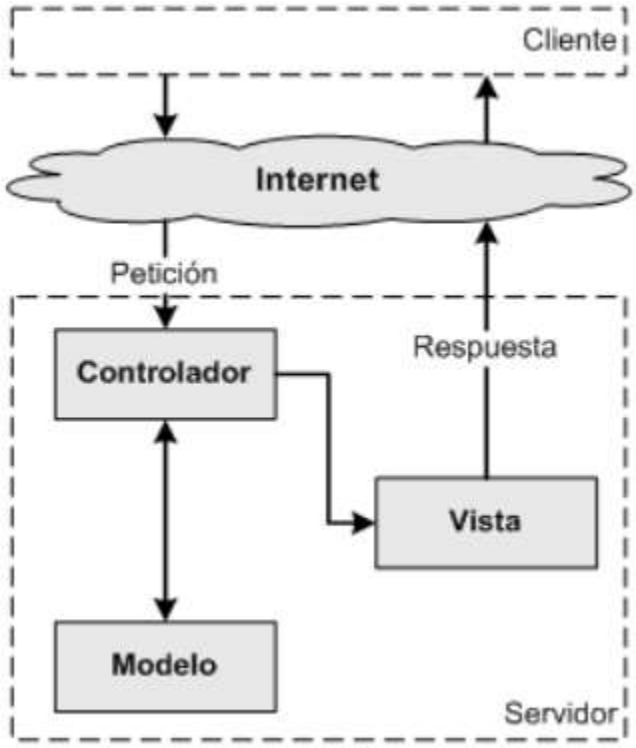
Estos últimos dos, cuando se utiliza el patrón Active Record para dicha capa.

2. La capa de Vista. Tiene las siguientes responsabilidades:
    a. Entrada y salida de datos del modelo.
    b. Permite al usuario realizar acciones que se convierten en eventos que serán manejados por la capa de control.
3. La capa Controlador. Tiene las siguientes responsabilidades:

a. Implementa los métodos que darán respuesta a los eventos disparados por el usuario.
b. Controla los flujos de ejecución de la aplicación: Invoca los servicios de la capa modelo, selecciona la vista con la que se dará respuesta a una petición, etc.

Estas tres capas están interconectadas entre sí, de forma que cada uno cumpla de manera correctas las responsabilidades que se le han delegado.

El siguiente diagrama describe el flujo de una petición para una aplicación Web con una arquitectura MVC. La petición es recibida por la capa de Controlador, y este determina si requiere invocar algún servicio de la capa Modelo. De ser así, la capa Modelo responde a la solicitud y posteriormente la capa Controlador invoca a la vista que corresponde y esa vista es la que se envía como respuesta al cliente.

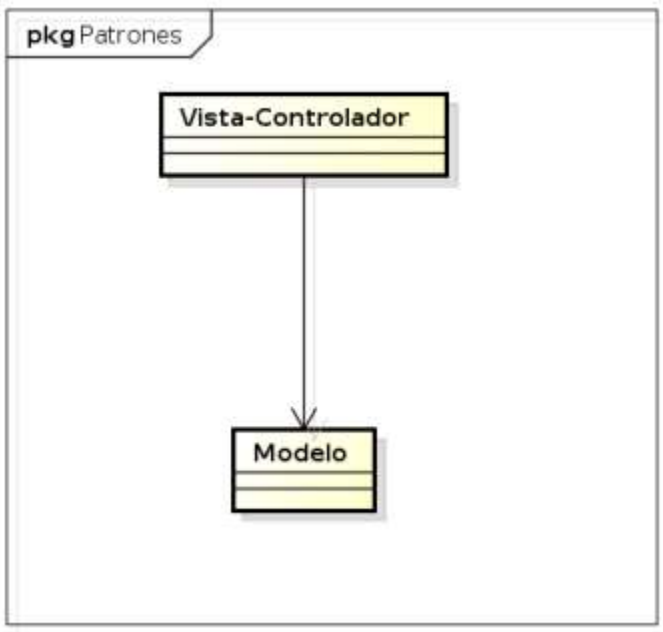


Usualmente los componentes de la vista no son clases, sino archivos con código HTML o alguna variante de lenguaje de marcado que posteriormente será traducido a código HTML.

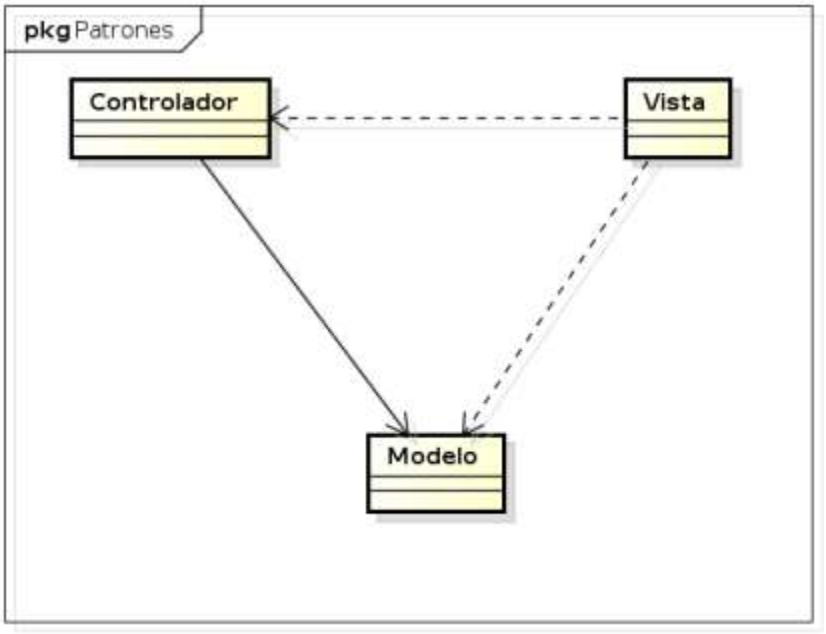
No obstante, dependiendo de la tecnología con que se desarrolle, el lenguaje de expresión (scriptlet) que se incluye en la vista es compilado o interpretado como una clase, por esta razón es que es posible concebir el modelo como capas de clases.

Existe una variedad de implementaciones del patrón MVC, que va desde frameworks (marcos de trabajo) hasta una implementación sin dependencias.

Usualmente en las aplicaciones de escritorio (como por ejemplo las construidas con Java Swing), la implementación difiere en cuanto a qué, la capa vista también asume las responsabilidades de controlador. Esta variante del modelo es usualmente llamada Modelo Vista-Controlador (con guión).

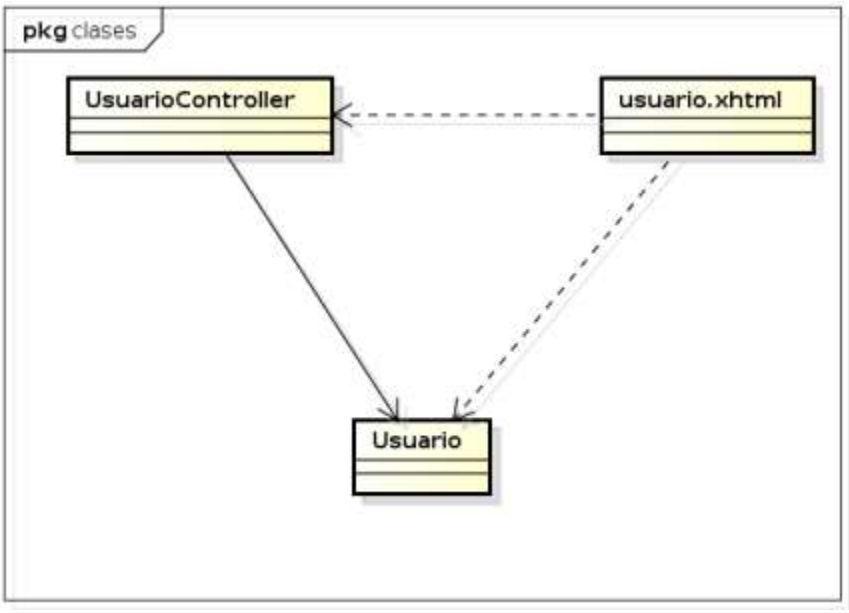


En las aplicaciones Web es necesario hacer una separación completa de las responsabilidades de las tres capas. La implementación más comúnmente realizada en entornos Web, es la que se describe a través del siguiente diagrama de clases.



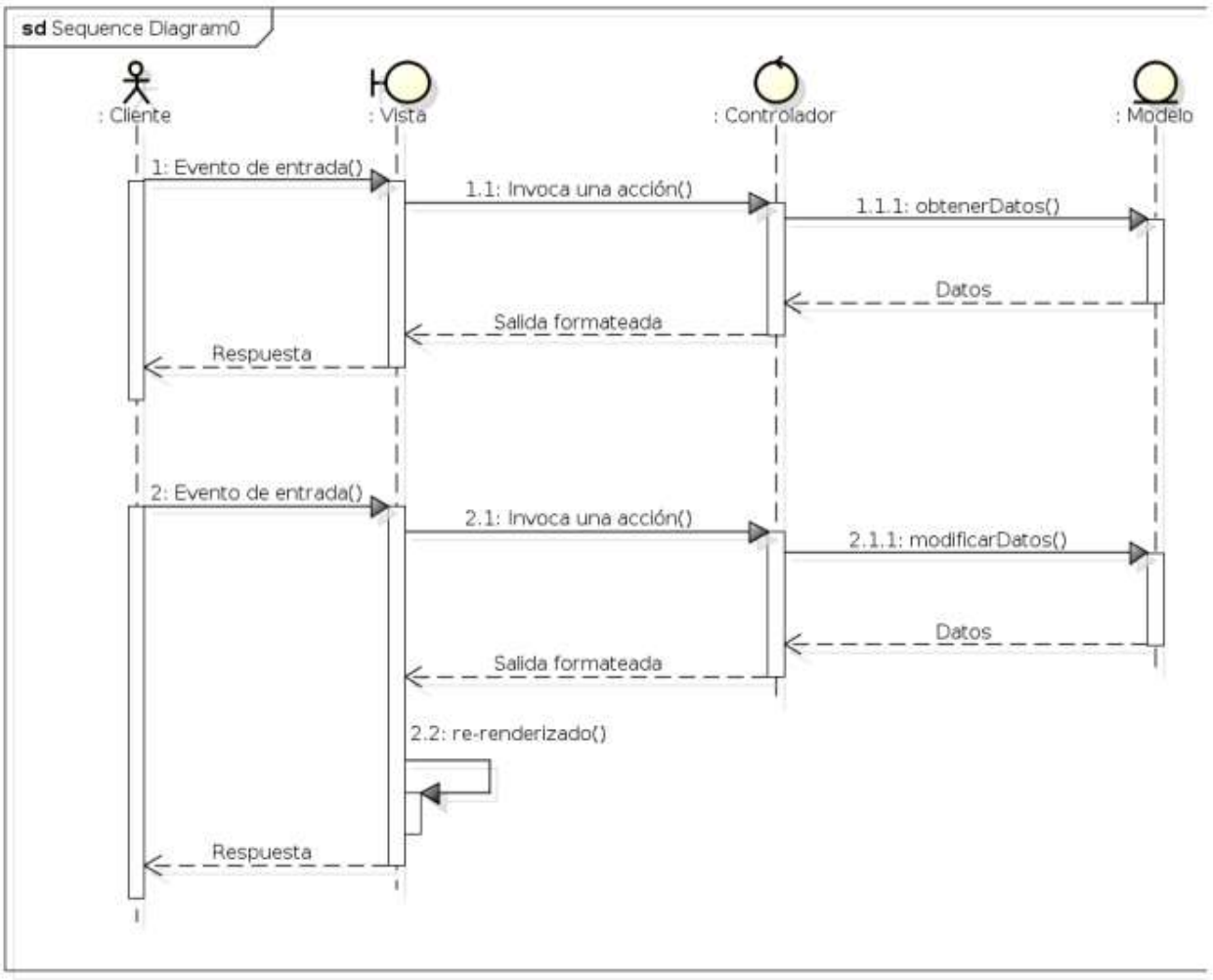
Esta implementación supone que la capa Vista usa objetos de la capa Controlador y de la capa Modelo. La capa Controlador invoca los servicios de la capa Modelo.

Este modelo es una descripción abstracta de cómo se relacionarán todas las clases de la aplicación. Así, los requerimientos de una aplicación que tiene que ver con la gestión de usuarios, tendrá una implementación concreta como se muestra en el siguiente diagrama de clases:



Donde la clase UsuarioController es la clases controlador, la página usuario.xhtml es la vista y la clase Usuario es clase de entidad que por ende pertenece a la capa modelo.

De igual manera, es posible especificar un comportamiento general para todas las peticiones que se realicen en un diseño MVC. El diagrama de secuencia que se muestra, describe cómo será la interacción entre los objetos de las capas, ante una petición del usuario.



El primer flujo corresponde a una petición de consulta, donde el evento disparado por el usuario es el de recuperar los datos de alguna entidad del modelo.

El segundo flujo, corresponde a una petición de modificación. El usuario podría haber modificado un objeto del modelo, creado uno nuevo o eliminado uno existente. Esto supone que la interfaz requiere ser renderizada nuevamente para que los cambios sean visualizados por el usuario.

Nuevamente, nótese que el diagrama de secuencia describe el comportamiento general (el patrón) de cómo se comportarán todos los objetos del sistema, así se trate de Productos, Usuarios, Clientes, Cuentas, etc.

De igual manera que con el diagrama de clases, podemos describir el comportamiento específico del flujo, por ejemplo, de una interfaz para administrar usuarios.

Además de resolver los problemas de mantenimiento y escalabilidad, el patrón MVC también presenta otras ventajas para los proyectos de desarrollo:

1. Especialización y separación de funciones de programación. Al separar las capas, es posible contar con especialistas en distintas tecnologías. Por ejemplo, podemos tener un programador especialista en tecnología del lado del cliente (HTML, XHTML, JavaScript y CSS) y un especialista en tecnología del lado del servidor (Java, PHP, C#, Python, etc.).
2. Desarrollo en paralelo. En la medida en que contamos con especialistas en distintas tecnologías, podemos organizar el proceso de desarrollo en forma de producción en serie. Esta producción en serie permitirá que un programador de interfaces pueda estar programando únicamente interfaces para más de un módulo o más de un sistema al mismo tiempo. Al igual que el programador de clases del negocio.
3. Reusabilidad. En la medida en que el software ha sido separado en capas, es posible reutilizar algunas de estas capas por separado, no solamente en la misma aplicación, sino en otras aplicaciones donde se requiera.