LABORATORIO DE PATRONES DE ARQUITECTURA: CAPAS Y MVC, MICROKERNEL

OBJETIVO

Aplicar los patrones Capas y MVC, así como el patrón de arquitectura microkernel para el diseño de sistemas software flexibles y fáciles de modificar.

BASE CONCEPTUAL

1. PATRÓN CAPAS

Nombre del patrón: Capas

Intención del patrón: el patrón de arquitectura Capas se utiliza para estructurar un sistema en múltiples niveles de abstracción, donde cada capa se encarga de un aspecto específico de la funcionalidad del sistema. Esto promueve la modularidad, la reutilización y la mantenibilidad del software aplicando el principio de separación de preocupaciones y establece una clara jerarquía de responsabilidades y de uso entre las diferentes capas.

Tipo de patrón: Módulos (Los módulos son capas, la relación autorizada a usar, se pierde la descomposición recursiva, una capa no se descompone en capas sino en segmentos).

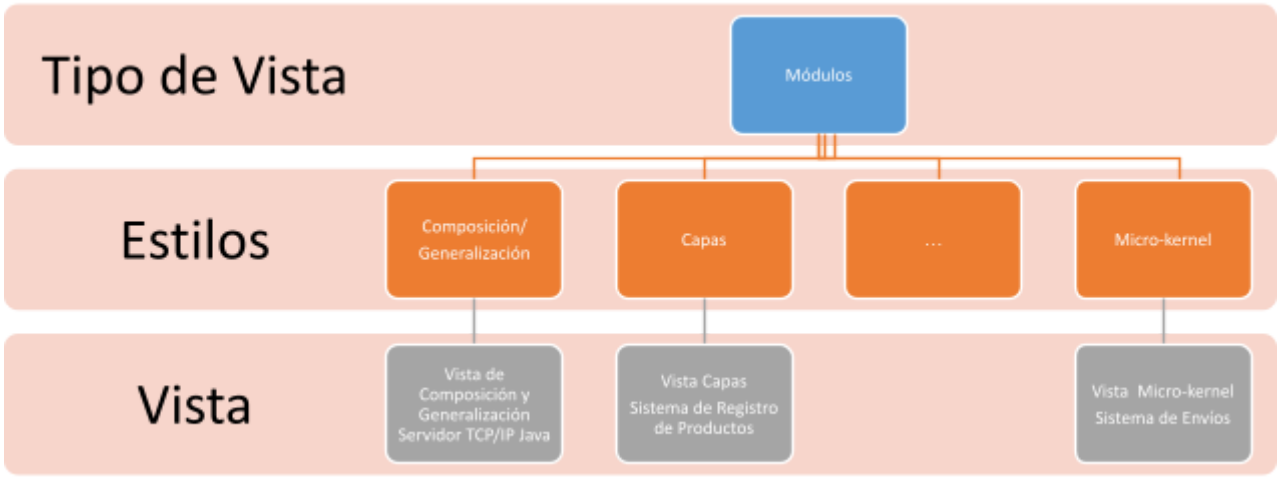

Figura 1. Patrón capas como estilo arquitectónico para generar vistas de tipo módulos

Problema: se desea diseñar un sistema que sea fácil de mantener, extender y comprender, donde las distintas partes del sistema estén claramente definidas y separadas según su función y nivel de abstracción. Además, se busca facilitar la reutilización de componentes y la integración de nuevas funcionalidades sin afectar el resto del sistema.

Motivación: el patrón de diseño Capas se inspira en la organización de sistemas complejos, como aplicaciones empresariales o sistemas de información, donde es crucial mantener una separación clara entre la presentación, la lógica de negocio y el acceso a datos. Al dividir el sistema en capas, se facilita la gestión de cambios y la evolución del software durante su mantenimiento.

Estructura del patrón: el patrón de diseño arquitectónico en capas consta de varias capas, cada una con un propósito específico y una responsabilidad claramente definida. Las capas suelen incluir la capa de presentación (interfaz de usuario), la capa de aplicación (lógica de negocio), la capa de acceso (persistencia u otros sistemas) y, en algunos casos, capas adicionales como la capa de servicios o la capa de dominio.

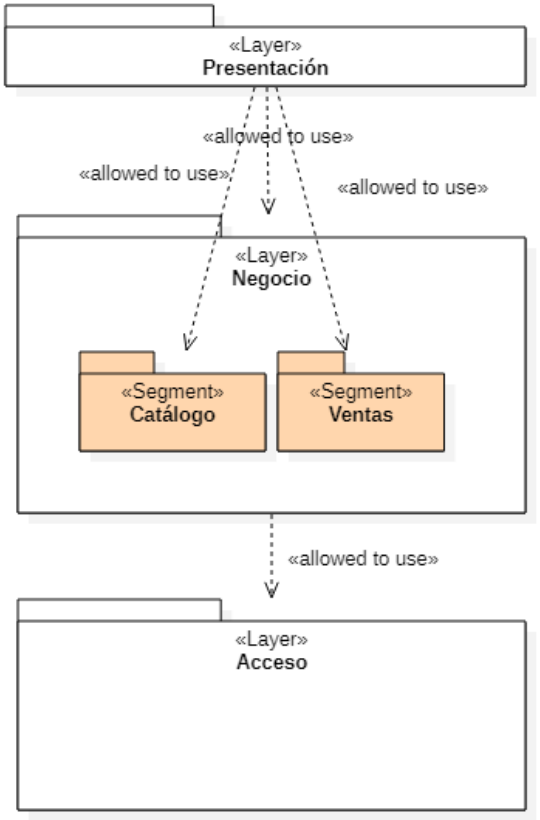

Figura 2. Estructura y ejemplo de uso del patrón capas. Observar los estereotipos para los módulos Layer, las dependencias allowed to use y la composición de capas por segmentos (segment)

Modelo computacional: Máquina Virtual, una abstracción que permite ejecutar múltiples instancias de un sistema operativo y sus aplicaciones en un único hardware físico. Consiste en emular una plataforma de hardware completa dentro de otra, de modo que el sistema operativo y las aplicaciones que se ejecutan dentro de la máquina virtual interactúen con esta plataforma virtual como si estuvieran funcionando en un entorno físico independiente. Este mismo concepto escalado a nivel software es lo que se homologa a una capa.

Ejemplo:

Considere un sistema de venta para una tienda en línea. Este sistema podría estar dividido en varias capas: la capa de presentación, que incluye la interfaz de usuario para que los usuarios realicen pedidos; la capa de negocio, donde se encuentra la lógica para procesar gestionar los productos y procesar las ventas; la capa de acceso, que se encarga de interactuar con la base de datos para almacenar y recuperar la información de los productos y pedidos. Cada capa tendría sus propios componentes y responsabilidades, lo que facilitaría el mantenimiento y la evolución del sistema a medida que cambien las necesidades de la organización y del negocio.

Consecuencias

Separación de Responsabilidades: una clara separación de responsabilidades entre las diferentes capas del sistema, facilitando la comprensión, el mantenimiento y la evolución del software, ya que cada capa se centra en un aspecto específico de la funcionalidad del sistema.

Modularidad, mantenibilidad, facilidad de prueba y reutilización: La división del sistema en capas promueve la modularidad al permitir que cada capa se desarrolle, pruebe y mantenga de forma independiente. Esto facilita la reutilización de componentes entre proyectos y la integración de nuevas funcionalidades sin afectar otras partes del sistema.

Estandarización del desarrollo: una arquitectura en capas fomenta la estandarización y la consistencia en el diseño y la codificación del software. Esto facilita la colaboración durante el desarrollo, ya que todos siguen una estructura común con fronteras bien definida para la construcción del sistema entre varios equipos.

Aunque el patrón en capas ofrece los beneficios anteriores, aplicarlo tiene algunas limitaciones:

Desempeño: agregar capas en un sistema puede tener un impacto negativo en los tiempos de procesamiento, ya que cada capa introduce una sobrecarga adicional de procesamiento y comunicación. Esto puede ser especialmente problemático en sistemas con requisitos de desempeño muy estrictos como los sistemas de tiempo real.

Exceso de acoplamiento: Si las capas están demasiado acopladas entre sí, puede ser difícil realizar cambios en una capa sin afectar a las otras capas del sistema. Esto puede limitar los beneficios que promete el patrón en términos de modificabilidad y capacidad de prueba.

Usos conocidos:

El Modelo OSI - Open Systems Interconnection, es un modelo de referencia propuesto por la Organización Internacional de Normalización (ISO) para estandarizar y describir los procesos de comunicación de datos entre sistemas de computadoras en una red. El modelo OSI se compone de siete capas, cada una con un conjunto específico de funciones y responsabilidades. Otro ejemplo son los sistemas de gestión de recursos empresariales (ERP), que normalmente se han construido por capas considerando la capa de presentación para la interfaz de usuario, una capa de negocio para la lógica empresarial y una capa de acceso a datos para acceder a la base de datos. Ver Figura 2.

2. PATRON MICROKERNEL

Nombre del patrón: Microkernel

Intención del patrón: Se basa en la idea de mantener un núcleo mínimo y mover la funcionalidad adicional al espacio de usuario mediante módulos independientes. Esto promueve la modularidad, la flexibilidad y la capacidad de personalización del sistema, al tiempo que mejora algunos aspectos de confiabilidad como la seguridad y la tolerancia a fallos.

Tipo de patrón: Módulos (Diagramas de paquetes y diagramas de componentes de implementación)

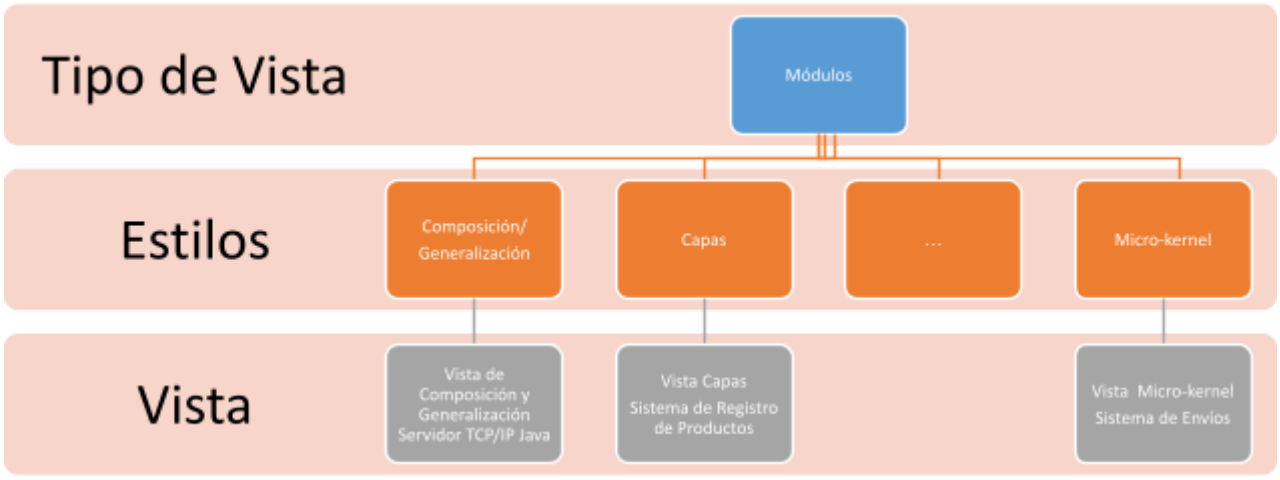

Figura 3. Patrón microkernel como estilo arquitectónico para generar vistas de tipo módulos

Problema

Se desea construir un sistema personalizable, comercializable por partes o una línea de productos en un contexto de negocio en que se quiere abarcar varios mercados y reutilizar de manera planificada la solución con el fin de lograr incrementar la calidad, la productividad y el tiempo de salida al mercado.

Motivación

Un sistema operativo como Linux que está basado en el patrón microkernel, el núcleo proporciona un conjunto mínimo de servicios esenciales, como la gestión de la memoria, la planificación de procesos y la comunicación entre procesos. Estos servicios básicos se implementan de manera eficiente y se mantienen lo más pequeño, independiente y simple posible.

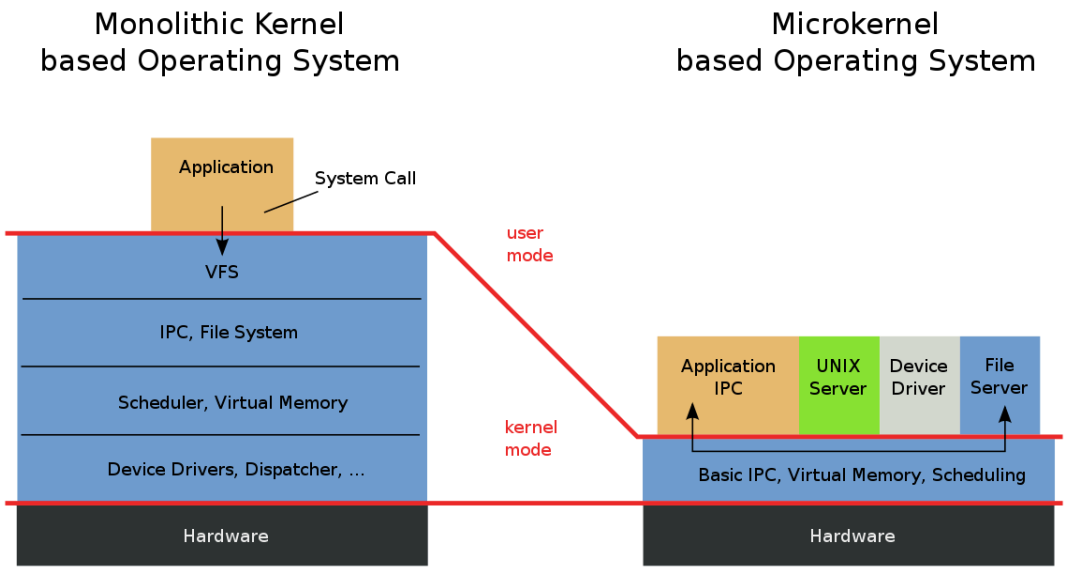

Figura 4. Control de llamados en un enfoque monolítico versus basado en microkernel para el diseño de un sistema operativo

Por otro lado, las funcionalidades adicionales, como los sistemas de archivos, los controladores de dispositivos y las interfaces de red, se implementan como módulos independientes que se ejecutan en espacio de usuario y se comunican con el núcleo a través de interfaces definidas. Estos módulos pueden cargarse o descargarse dinámicamente, lo que permite la extensibilidad y la capacidad de personalizar el sistema según las necesidades específicas.

Estructura del Patrón

El patrón de arquitectura microkernel es un enfoque de diseño para sistemas de software que busca minimizar las dependencias del sistema de tal forma que se construye un pequeño ***núcleo o kernel*** del sistema, delegando las funcionalidades variables a través de módulos (***plug-in***) que se agregan y configuran en una aplicación específica.

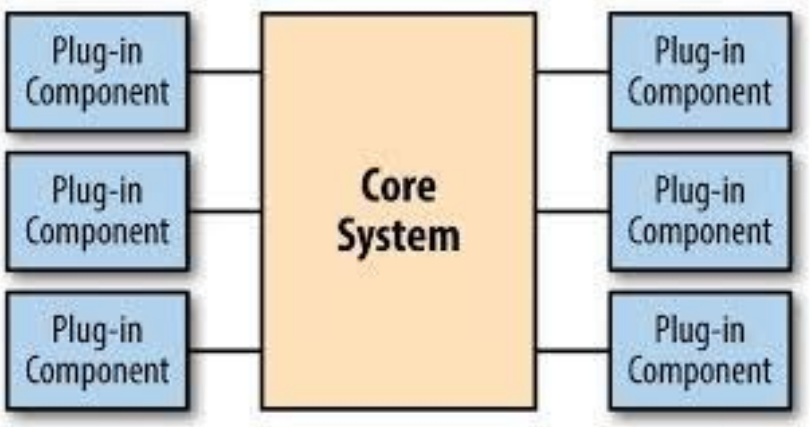

Figura 5. Arquitectura Conceptual del Patrón Microkernel

Para ello el núcleo trabaja bajo el principio de inversión de dependencias, dependiendo de un ***manejador*** de plugins y de las ***intefaces*** establecidas para el uso de los plugins por parte del ***núcleo o kernel***.

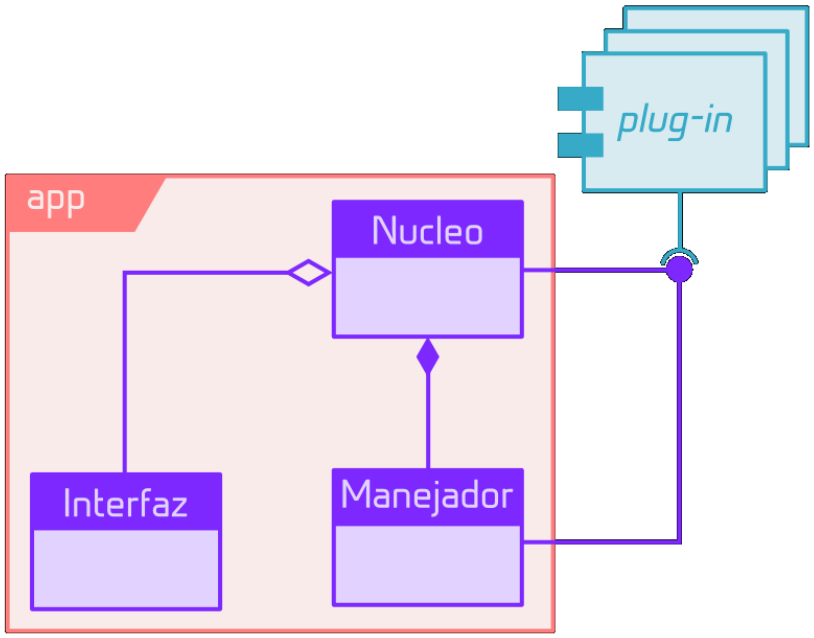

Figura 6. Arquitectura Lógica del Patrón Microkernel

Modelo computacional: Sistema reflexivo, el cuál es un sistema que puede observar y modificar su propia estructura y comportamiento durante su ejecución. Esta capacidad de introspección y reflexión permite que el sistema se adapte dinámicamente a cambios en su entorno sin necesidad de intervención humana o de otro sistema.

Ejemplo

Se tiene un sistema de envío de productos comprados, una aplicación de logística tiene la funcionalidad de **calcular el costo de envío** de paquetes. Por ahora, solo opera para algunos países como México, Colombia y Chile. Sin embargo, tiene planes muy cercanos de expansión en Latinoamérica. El siguiente diagrama de módulos describe la solución general basada en microkernel. En el núcleo se encuentra la lógica de negocio, la lógica de presentación se ha omitido. La lógica de gestión (manejador) de plugins se ha incluido en el núcleo (PluginManager). De forma independiente (en otros módulos) se ha definido la interface de los plugins y los plugins. La figura 6 presenta la forma en que se extiende un plugin (Clase MexicoDeliveryPlugin) a partir de la interface de plugin definida (DeliveryPlugin). En la vista 7 pueden verse los componentes de implementación (“proyectos” diferentes) conectados a través de Maven considerando las dependencias de tiempo de compilación y de tiempo de ejecución.

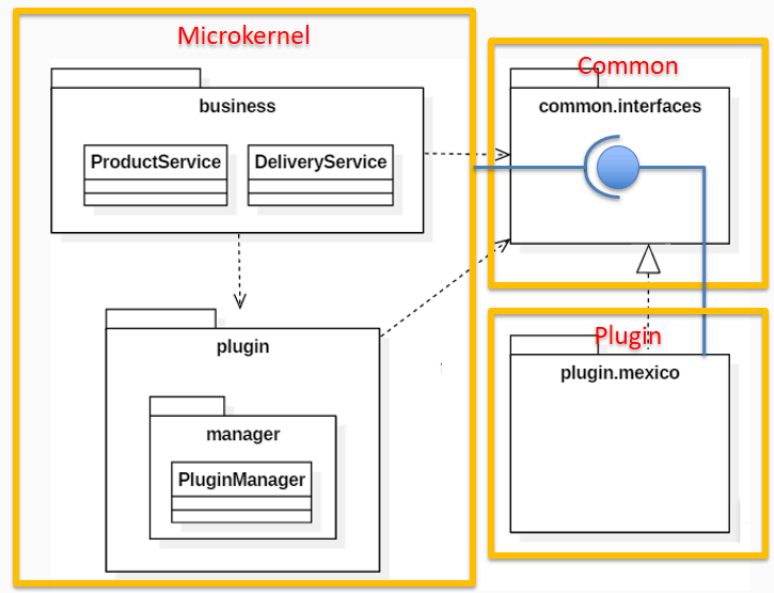

Figura 7. Vista de Módulos del sistema de logística de envío

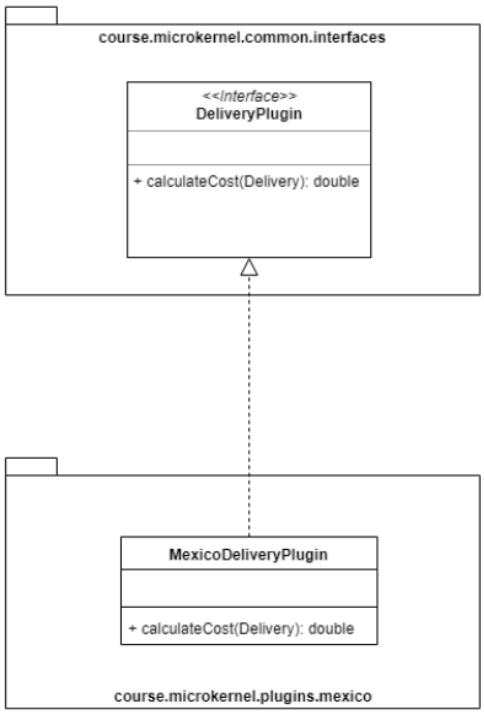

Figura 8. Extendiendo un Plugin para el cálculo de costo de envío para México

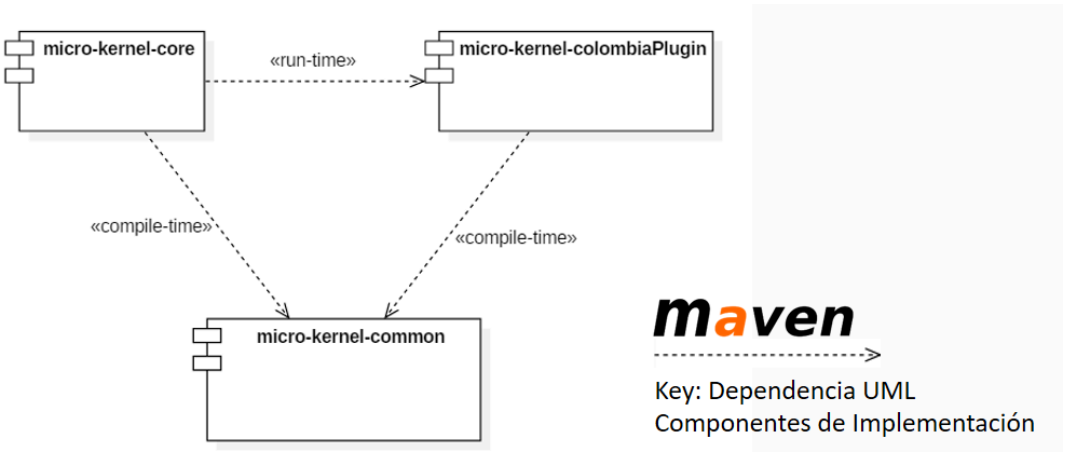

Figura 9. Vista de Implementación (Componentes de implementación) con el plugin de Colombia

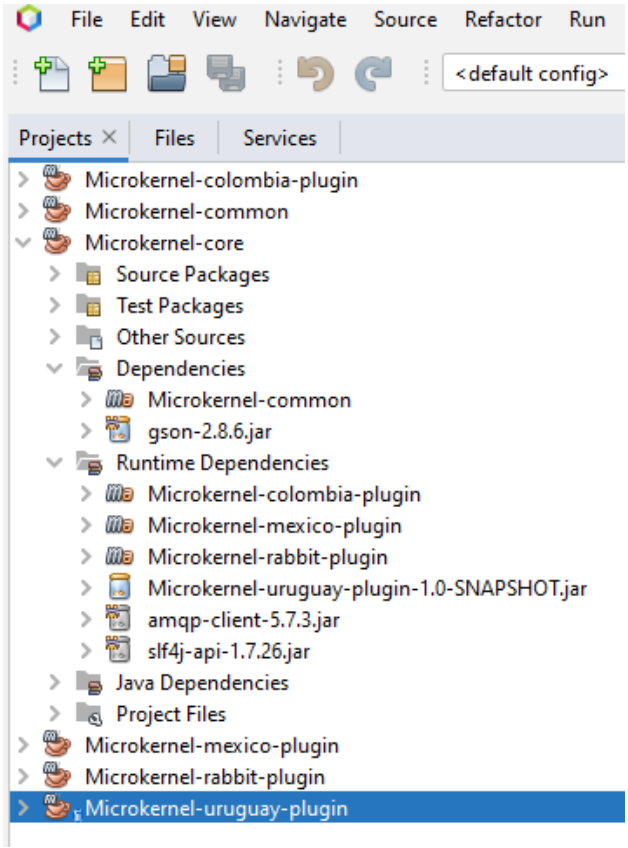

Figura 10. Vista desde el ambiente de desarrollo Netbeans donde se pueden ver las dependencias del núcleo respecto al componente commons y a los plugins (Colombia, México y Uruguay)

Código del ejemplo

Se adjunta código en el taller

Consecuencias del uso de patrón

El principal beneficio del enfoque microkernel es la modularidad y la flexibilidad que proporciona. Al mantener un núcleo mínimo y mover gran parte de la funcionalidad a la personalización de la solución, se facilita la incorporación de nuevas características o la modificación de las existentes sin tener que modificar el núcleo central del sistema. Esto mejora la capacidad de mantenimiento y la robustez del sistema. Además, la separación de los servicios del núcleo también contribuye a la confiabilidad: si un módulo plugin falla, no afectará directamente al núcleo y, por lo tanto, el sistema puede recuperarse o seguir funcionando sin necesidad de reiniciar por completo.

Usos conocidos

RabbitMQ permite agregar funcionalidades adicionales mediante el uso de plugins. RabbitMQ está construido utilizando la arquitectura de microkernel. En el caso de RabbitMQ, el núcleo del sistema se encarga de las tareas esenciales relacionadas con la mensajería, como el enrutamiento y la entrega de mensajes. Sin embargo, muchas funcionalidades adicionales, como la autenticación, el cifrado, la gestión de colas y el enrutamiento avanzado, están disponibles a través de plugins. Estos plugins se pueden agregar o quitar según las necesidades del usuario, lo que permite ampliar la funcionalidad de RabbitMQ de manera modular. Esto hace que RabbitMQ sea altamente flexible y adaptable a diferentes casos de uso y requisitos específicos de los sistemas de mensajería.

3. El Patrón MVC

El patrón MVC (Modelo-Vista-Controlador) es un patrón de diseño de la micro-arquitectura que separa una lógica de la presentación (IU) en tres componentes principales: el Modelo, la Vista y el Controlador.

El Modelo representa los datos que se obtienen de la lógica de negocio o la lógica de negocio misma. El componente modelo encapsulan el estado y el comportamiento de la aplicación. Los componentes modelo son responsables de mantener la integridad de los datos y de notificar a los componentes de vistas sobre los cambios en el estado de los datos mediante una invocación implícita (usando el patrón de observador). Los modelos pueden ser simples representaciones de datos o pueden contener lógica de negocio más compleja, dependiendo de las necesidades de la aplicación.

La Vista: este componente es el responsable de mostrar en la interfaz de usuario la información al usuario y también capturar eventos de su interacción. Las vistas pueden ser ventanas, widgets, gráficos u otras representaciones visuales de la información. Las vistas se mantienen actualizadas mediante la

observación de los cambios en el modelo. Cuando el modelo cambia, notifica a las vistas asociadas, lo que les permite actualizar su representación visual para reflejar los cambios en los datos.

El Controlador: este componente actúa como intermediario entre los componentes Modelo y la Vista. Los controladores son objetos que manejan las interacciones del usuario y coordinan las acciones entre el modelo y la vista. Los controladores interpretan los eventos de entrada del usuario (vienen de la vista en forma implícita ej. Listeners en Java), realizan operaciones en el modelo según esas entradas y mueven la interacción del usuario de una vista a otra si es necesario. Los controladores suelen ser objetos que responden a eventos de usuario, como clics de ratón o pulsaciones de teclas, y que desencadenan acciones en el modelo o cambian de vista en función de esas interacciones.

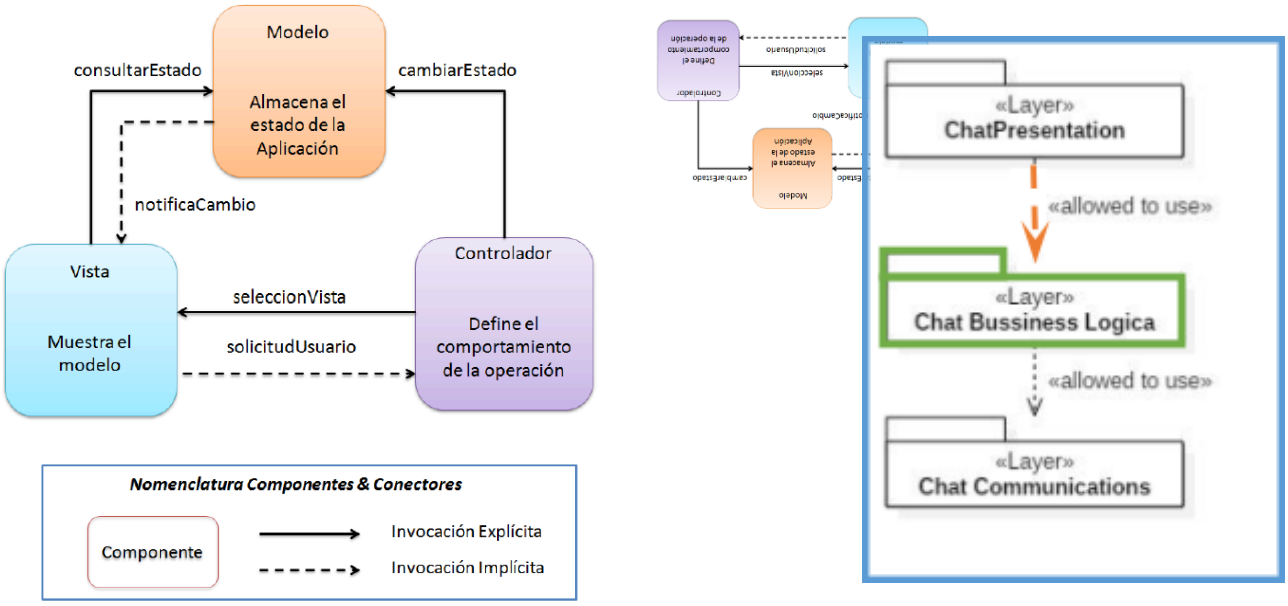

Figura 11. Patrón MVC al lado izquierdo las interacciones. Al lado derecho su relación con el patrón de arquitectura Capas.

El patrón MVC usa el patrón Observador, para más información ir al documento:
http://artemisa.unicauca.edu.co/~ahurtado/ingsoftware/PatronesdisenoDoc.pdf

DESCRIPCIÓN DEL TALLER

En el anterior taller organizaron y mejoraron un sistema que permite la gestión productos y sus categorías. Agregar la funcionalidad de carrito de compra a este sistema con los siguientes requisitos:

a. Se debe agregar una ventana simple para agregar productos a la lista de compras. Es la misma ventana de buscar por nombre, se selecciona el producto y se le da al botón ***agregar al carrito de compras***. El sistema le despliega una ventana con la información del producto, una caja de texto para indicar la cantidad y la posibilidad de agregar o cancelar este ***ítem de compra***.

b. Se debe tener otra ventana que actualiza el carrito de compras debe ser actualizada cada vez que se agrega un ítem a la compra de manera automática (por notificación aplicando MVC). El carrito de compras es una ventana "observadora" que tiene los ítems de compra que corresponde a la dupla producto y cantidad, así como el total de la compra. Además, tiene la posibilidad de cancelar la compra o proceder al pago.

c. Para el pago se desea usar diferentes plataformas de pago. Debe implementarse una forma de pago simulada llamada *PaySimulated*, abriendo un archivo (que tiene un código y un saldo en forma encriptada, con el código se valida y el saldo se cambia en caso de hacerse la compra). Sin embargo, la idea es que la solución se diseñe e implemente para varias plataformas de pago a través de la instalación de un plugin correspondiente (PayPal, GooglePay, Stripe, entre otras), por lo que se requiere en este requerimiento abordar el patrón de microkernel.

La entrega debe realizarse de la siguiente forma:

1. Crear una carpeta y dentro colocar el archivo *integrantes.txt* con los integrantes del proyecto y la URL del código fuente del repositorio github resultado de implementar los requisitos a, b y c.
2. Se debe entregar el diagrama de módulos siguiendo el patrón de capas (module_view.png) y un Diagrama Clases que detalle las clases y relaciones que surgieron del diseño detallado (class_diagram.png)
3. Comprimir la carpeta y enviarla en la tarea respectiva de univirtual.unicauca.edu.co.