
**Universidad Nacional**
**Autónoma de México**


**Facultad de Ciencias**

**Ciencias de la Computación**
**Modelado y Programación**
**Proyecto 01 - 2025-2**

**Equipo: Chocolates Turing**
**Integrantes:**


- Labrador Mata Janet 321126953
- Luna Villanueva Karla Victoria 321024189
- Hernández Islas Leonardo Daniel 321279808


**Profesora: Rosa Victoria Padilla**
**Fecha de entrega: 21 de abril del 2025**

## 1. Clases principales:

- Componente
- CPU
- RAM
- HDD
- SSD
- FuentePoder
- Motherboard
- Gabinete
- Computadora
- Compatibilidad
- SoftwareAdicional
- Sucursal
- Disc
- Main

## 2. Interfaces:

- Compatibilidad
- ComponenteFactory
- EnsamblajeEstado
- EstadoEnvio

## 3. Componentes:

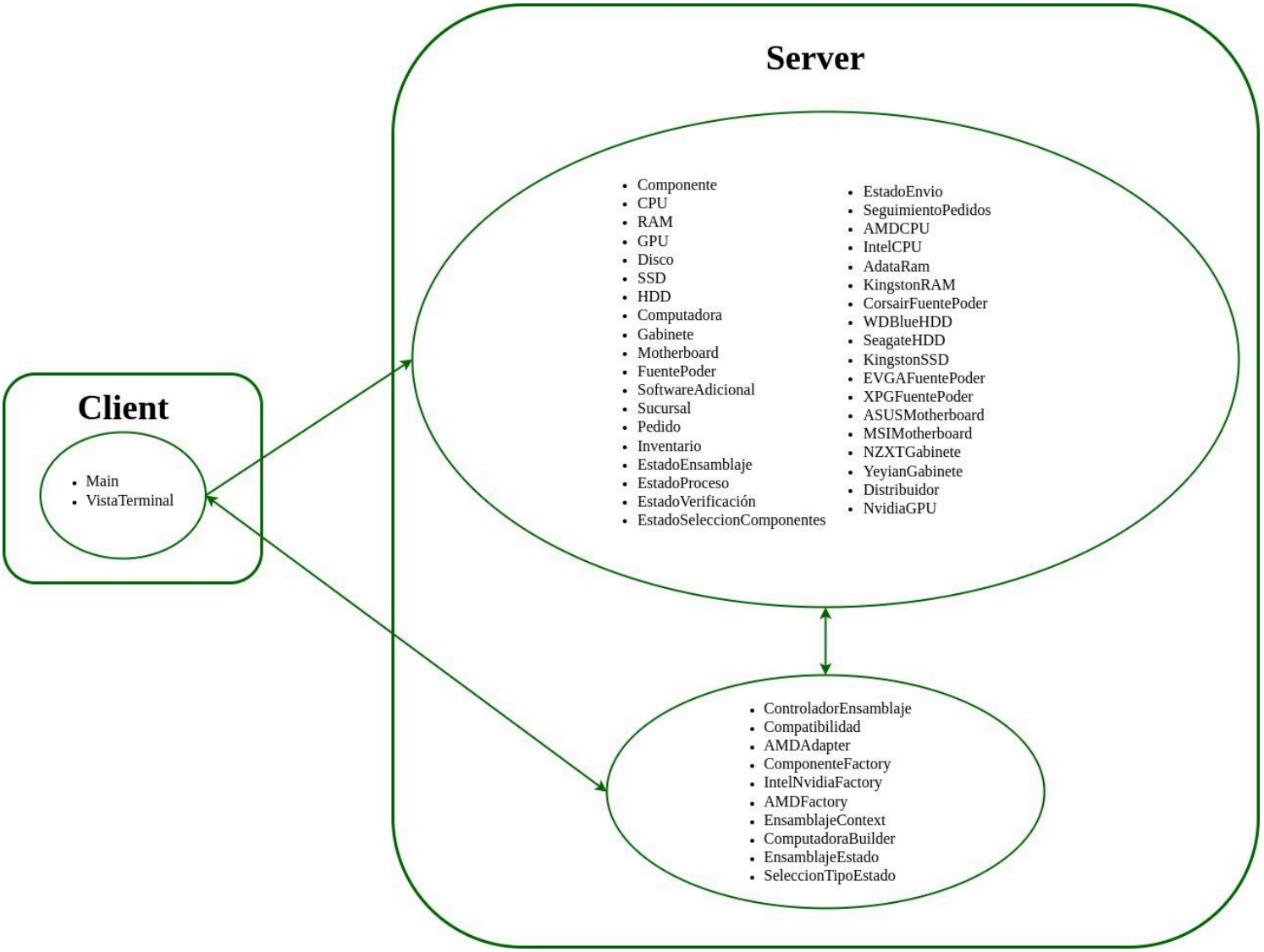
Los componentes que maneja actualmente la empresa son los siguientes:

- **Procesador (CPU)**
  - Intel
    - Core i3-13100 **$1200.00**
    - Core i5-13600K **$2500.00**
    - Core i7-13700K **$3500.00**
    - Core i9-13900K **$5000.00**
- **RAM**
  - Adata
    - 8 GB **$800.00**
    - 16 GB **$1500.00**
    - 32 GB **$2800.00**
  - Kingston
    - 8 GB **$850.00**
    - 16 GB **$1600.00**
    - 32 GB **$3000.00**
- **Motherboard**
  - ASUS
    - ROG Maximus Z790 Hero **$5000.00**
    - TUF Gaming B760-Plus WIFI D4 **$3200.00**
  - MSI
    - MEG Godlike **$6000.00**
    - MAG B760 Tomahawk WIFI DDR4 **$3500.00**
- **HDD (Discos Duros Mecánicos)**
  - Western Digital Blue
    - 500 GB **$700.00**
    - 1TB **$1000.00**
  - Seagate Barracuda

        - 1TB **$1100.00**
        - 2TB **$1600.00**
- **SSD (Discos de Estado Sólido)**
    - Kingston
        - 500 GB **$1200.00**
        - 1 TB **$1800.00**
        - 2 TB **$2500.00**
        - 4 TB **$4000.00**
- **Fuente de alimentación**
    - EVGA
        - 800 W **$1000.00**
        - 1000 W **$1300.00**
        - 1500 W **$1800.00**
    - Corsair
        - 800 W **$1050.00**
        - 1200 W **$1600.00**
        - 1500 W **$1900.00**
    - XPG
        - 500 W **$1000.00**
        - 700 W **$1300.00**
        - 1000 W **$1900.00**
- **Tarjeta gráfica (GPU)**
    - NVIDIA
        - GTX 1660 **$2500.00**
        - RTX 3060 **$3000.00**
        - RTX 4070 **$4500.00**
        - RTX 4080 **$6000.00**

        - RTX 4090 **$8000.00**
- **Gabinetes**
    - NZXT
        - H6 Flow ATX **$2000.00**
    - Yeyian
        - Lancer ATX **$1800.00**

## 4. Arquitectura MVC:



- **Client:**

  -Main, VistaTerminal:

  Estas clases están relacionadas directamente con la interacción del usuario pues muestran menús, despliegan resultados, presentan opciones y reciben elecciones.

- **Server:**

  -Model(CPU, RAM, GPU, Disco, SSD, HDD, Computadora, Gabinete, Motherboard, FuentePoder, SoftwareAdicional, Sucursal, Pedido, Inventario, EstadoEnsamblaje, EstadoProceso, EstadoVerificación, EstadoSeleccionComponentes, EstadoEnvio, SeguimientoPedidos, AMDCPU, IntelCPU, AdataRAM, KingstonRAM, WDBlueHDD, SeagateHDD, KingstonSSD, CorsairFuentePoder, EVGAFuentePoder, XPGFuentePoder, ASUSMotherboard, MSIMotherboard, NZXTGabinete, YeyianGabinete, Distribuidor, NvidiaGPU):

  Las clases anteriores modelan directamente la estructura del negocio junto con sus datos, ya que representan componentes físicos, procesos y estados del sistema.

  -Controller(ControladorEnsamblaje, Compatibilidad, AMDAdapter, ComponenteFactory, IntelNvidiaFactory, AMDFactory, ComputadoraBuilder, EnsamblajeContext, EnsamblajeEstado, SeleccionTipoEstado):

Estas clases controlan la lógica del flujo, y la interacción entre el View y el Model, es decir, conecta lo que el usuario quiere hacer con lo que el sistema puede hacer.

## 5. Patrón Adapter:

El proyecto está diseñado para trabajar con componentes de tipo Intel como lo es en el caso de CPU, cuando se intenta usar componentes de tipo AMD se necesita que estos mismos tengan un comportamiento compatible con la lógica inicial del sistema; en este caso AMDAdapter implementa la misma interfaz que un componente de tipo Intel, dentro del adaptador se adapta la funcionalidad de AMD a la que el sistema espera, la clase Compatibilidad es la encargada de decidir cuando aplicar dicho adaptador según lo que prefiera el usuario. Gracias a este patrón componentes de tipo AMD se usan de forma transparente en el sistema como si fueran Intel, sin necesidad de reescribir todo el flujo, también tenemos mayor flexibilidad en el código para poder extender la compatibilidad sin alterar el código original.

## 6. Patrón Abstract Factory:

Este patrón permite la creación de familias de objetos relacionados sin acoplar código a clases concretas, para este caso nos permite crear componentes de distintas familias, la interfaz ComponenteFactory define métodos como crearGPU(), crearRAM(), entre otros, de esta manera las fábricas: IntelNvidiaFactory y AMDFactory, implementan dichos métodos para poder crear los componentes específicos. Gracias a

este patrón podemos armar computadoras completas con componentes consistentes entre sí sin condicionales múltiples, además desacopla el proceso de creación de objetos y facilita el soporte de nuevas arquitecturas creando nuevas fábricas.

## 7. Patrón Builder:

Notemos que la clase Computadora tiene múltiples atributos y configuraciones, por lo que crear una computadora de manera directa usando un constructor podría generar inflexibilidad en nuestro código; por eso existe la clase ComputadoraBuilder en la cual hay métodos que permiten agregar los componentes de una computadora para después crear una instancia final del objeto de tipo Computadora. Gracias a este patrón podemos construir computadoras de manera clara, paso a paso, y totalmente personalizadas por el usuario, además hay una separación clara entre la construcción y la representación del objeto y se reutiliza el mismo builder para crear múltiples variantes.

## 8. Patrón State:

Una computadora pasa por diferentes etapas durante su ensamblaje, cada etapa tiene sus características específicas, si usáramos if o switch para manejar lo anterior, entonces tendríamos un código más difícil de mantener; en este caso las clases EstadoEnsamblaje, EstadoEnvio, EstadoProceso, EstadoSeleccionComponente y EstadoVerificacion son las clases que representan dichas etapas o estados, todas ellas implementan la interfaz EnsamblajeEstado y EnsamblajeContext mantiene

una referencia al estado actual e invoca su comportamiento; cada estado implementa su propia lógica según la etapa del ensamblaje. Gracias a este patrón el comportamiento del sistema cambia automáticamente dependiendo del estado actual, cada estado maneja su lógica de forma encapsulada y podemos manejar el flujo del ensamblaje sin grandes condicionales, facilitando así el mantenimiento y la extensibilidad del código.

## 9. Patrón Singleton:

En sistemas complejos, hay clases que **deben tener una única instancia en todo el sistema**, ya que pueden representar un recurso compartido, si existieran múltiples instancias de estas clases, entonces podrían producirse inconsistencias en el código, el patrón Singleton nos asegura que sólo habrá una instancia de una clase y a su vez, que esta sea accesible globalmente desde cualquier punto del sistema. La clase Inventario, Distribuidor y SeguimientoPedidos implementa la estructura del patrón: cuentan con un atributo estático privado que almacena la única instancia que será compartida, tienen un constructor privado que impide que se creen nuevas instancias desde fuera de la clase y un método público estático que devuelve la única instancia existente, si no existe, la crea; veamos un pequeño análisis de las clases:

-Inventario: contiene un mapa con todos los componentes disponibles, se usa como fuente central para consultar o almacenar componentes, sin importar desde qué parte del

sistema se accede y solo hay un inventario general para evitar duplicados o falta de sincronización.

-Distribuidor: almacena la lista de sucursales disponibles donde se pueden enviar computadoras, además toda la lógica que involucra regiones, rutas o zonas de entrega depende de esta instancia central.

-SeguimientoPedidos: registra todos los pedidos y su estado actual, permite consultar pedidos por ID o sucursal, y actualizar su estado de entrega, además al ser Singleton, garantiza que los estados no se sobrescriban ni se repitan entre distintas vistas o procesos.

Gracias a este patrón se garantiza la consistencia de datos compartidos, evitamos la duplicación de recursos como listas, mapas y/o estados, y facilitamos el acceso a servicios centrales en cualquier parte del sistema.

## 10. Compilación y Ejecución:

**A la altura de la carpeta src**

**Para compilar:** javac *.java

**Para ejecutar:** java Main

## 11. Versión de Java:

**Java 17**