La **refactorización**, según se describe en el documento, es una técnica de la ingeniería del software que consiste en mejorar la estructura interna del código existente sin cambiar su comportamiento externo. Esto facilita la corrección de errores, la rápida comprensión del código y mejora la calidad general del software, permitiendo que se mantenga y extienda más fácilmente.

# Código limpio

1. **Nombres:**
    - Se enfatiza la importancia de usar nombres significativos y fáciles de buscar para variables, clases y funciones. Los nombres deben ser claros y descriptivos para facilitar la comprensión sin necesidad de comentarios adicionales.
2. **Funciones:**
    - Las funciones deben ser pequeñas y hacer una sola cosa para cumplir con el principio de responsabilidad única. Se desaconseja el uso de funciones grandes y se recomienda limitar la cantidad de argumentos para simplificar el mantenimiento y las pruebas.
3. **Polimorfismo:**
    - Se discute el uso del polimorfismo para manejar comportamientos similares entre diferentes tipos de objetos, promoviendo el uso de interfaces comunes para mejorar la flexibilidad y reducir la redundancia.
4. **Comentarios:**
    - Se advierte sobre los riesgos de depender demasiado de los comentarios para explicar el código. Los comentarios pueden quedar obsoletos y no deben sustituir a un código claro y legible.

5. **Objetos y Estructuras de Datos:**

- Se cubre la ley de Demeter, que promueve el bajo acoplamiento entre módulos, y el principio de encapsulamiento, que esconde los detalles de la implementación interna de los datos dentro de un objeto.

6. **Clases:**

    - Las clases deben ser pequeñas y enfocadas en un único propósito. Además, se debe seguir un orden lógico en la organización de las clases, empezando por constantes, seguido de variables de instancia, constructores, métodos de instancia y métodos de acceso.

7. **Principios SOLID:**

    - El documento concluye con una discusión sobre los principios SOLID, que son fundamentales para el diseño de software orientado a objetos, destacando la importancia de estructuras que permitan la extensión sin modificación, la segregación de interfaces, y la inversión de dependencias para un diseño más robusto y mantenible.

# SOLID

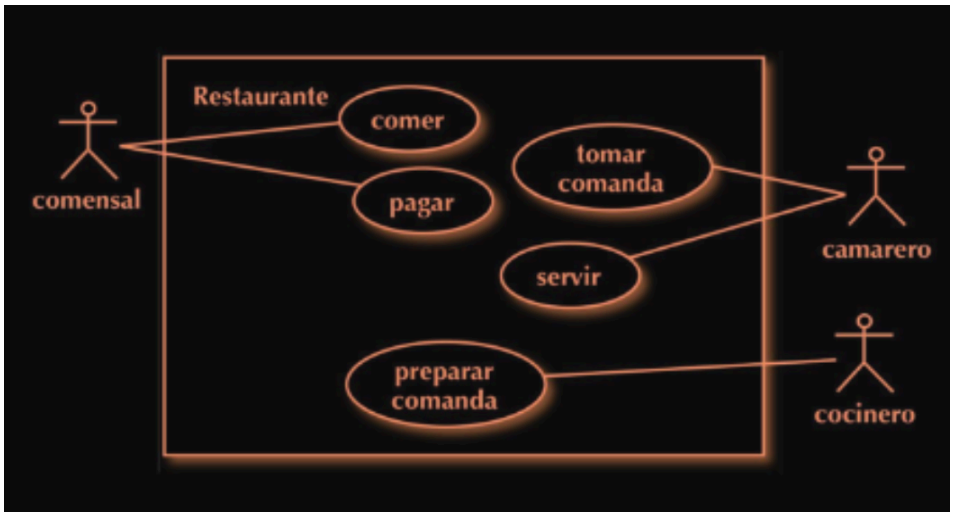
- **Principio de Responsabilidad Única (SRP):** Cada clase debería tener una sola razón para cambiar, encargándose de una sola parte de la funcionalidad.
- **Principio de Abierto/Cerrado (OCP):** Las entidades de software deben permitir la extensión de su comportamiento sin modificar el código existente.
- **Principio de Sustitución de Liskov (LSP):** Las subclases deben ser intercambiables con sus clases base sin alterar el comportamiento esperado del programa.
- **Principio de Segregación de Interfaces (ISP):** Diseña interfaces pequeñas y específicas para que los componentes no dependan de funcionalidades que no utilizan.
- **Principio de Inversión de Dependencias (DIP):** El diseño debe depender de abstracciones, no de concreciones, promoviendo un desacoplamiento del código.

**UML (Lenguaje de Modelado Unificado)** se describe como un lenguaje gráfico utilizado para visualizar, especificar, construir y documentar las partes que comprenden el desarrollo de sistemas de software.

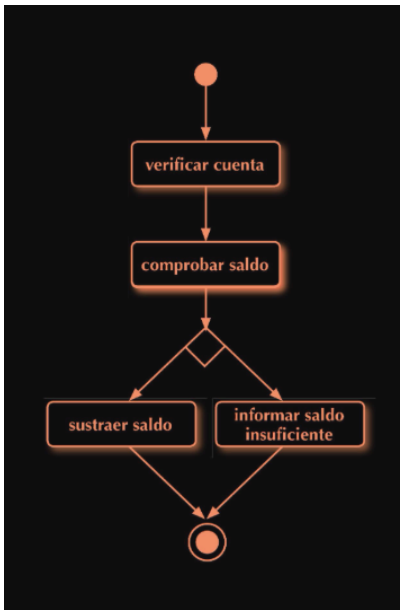
1. Componentes de UML:
    - **Diagramas de Clase:** Muestran las clases que componen el sistema y cómo se relacionan entre sí.
    - **Diagramas de Objeto:** Representan objetos específicos (instancias de clases) y sus relaciones en un momento dado.
    - **Diagramas de Caso de Uso:** Utilizados para entender cómo los usuarios interactuarán con el sistema, mostrando actores, acciones y relaciones entre ellos.

# Diagramas

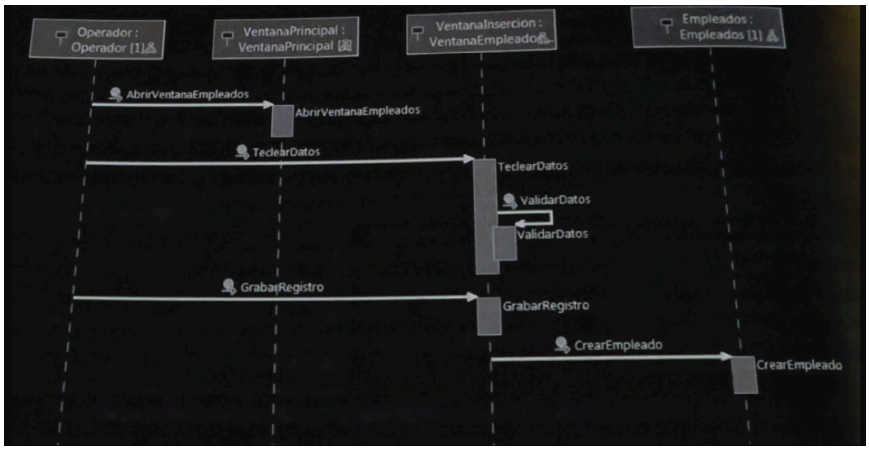
1. **Diagramas de Clase:** Ilustran las clases en un sistema, sus atributos, métodos y las relaciones entre las clases, incluyendo asociación, herencia, agregación y composición.
2. **Diagramas de Objeto:** Muestran instancias de clases (objetos) y sus relaciones en un momento específico, funcionando como una instantánea de la estructura de un sistema en un punto en el tiempo.
3. **Diagramas de Caso de Uso:** Ayudan a visualizar el funcionamiento del sistema desde la perspectiva de los usuarios externos, mostrando cómo los actores interactúan con el sistema a través de diversos casos de uso.



4. **Diagramas de Actividad:** Similar a los diagramas de flujo, estos diagramas muestran el flujo de actividades y las decisiones tomadas a lo largo de un proceso dentro del sistema.



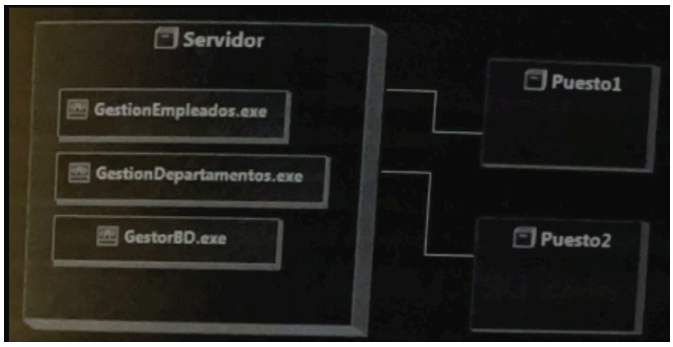

5. **Diagramas de Secuencia:** Representan la interacción entre objetos en el tiempo, mostrando cómo los objetos intercambian mensajes y en qué orden ocurren estas interacciones.



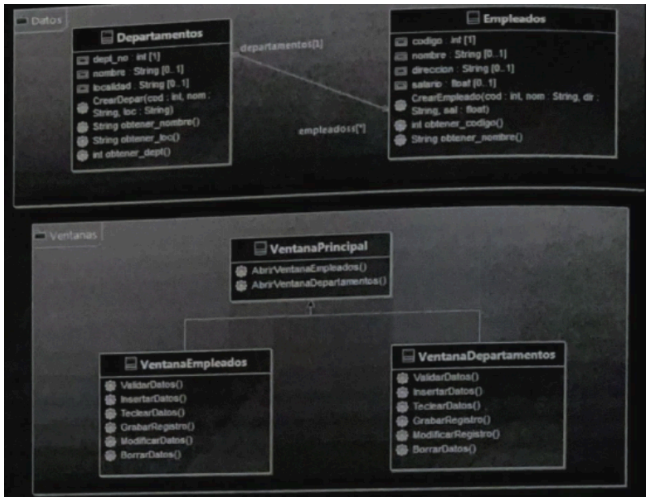
6. **Diagramas de Estados:** Utilizados para modelar los cambios de estado de un objeto en respuesta a eventos internos o externos, ilustrando cómo un objeto cambia de un estado a otro.

*os de un objeto Trabajador*

7. **Diagramas de Despliegue:** Especifican la configuración física del hardware y cómo el software se despliega y se ejecuta en ese hardware.




8. **Diagramas de Paquetes:** Organizan los modelos y elementos de UML en paquetes, que son agrupaciones de elementos relacionados, ayudando a estructurar un proyecto de desarrollo de software.



**Visibilidad de Atributos y Métodos:**

- **Public (+):** Visible tanto dentro como fuera de la clase.
- **Private (-):** Accesible solo desde dentro de la clase.
- **Protected (#):** Accesible dentro de la clase y por sus subclases.
- **Package (~):** Visible solo dentro del mismo paquete.

**Tipos de Relaciones:**

- **Asociación:** Conexión conceptual entre clases, puede ser bidireccional o unidireccional.
- **Herencia:** Relación jerárquica donde una clase hija hereda atributos y métodos de una clase padre.
- **Agregación:** Relación de "todo/parte" donde las partes pueden existir independientemente del todo.
- **Composición:** Relación de "todo/parte" más fuerte donde las partes no pueden existir sin el todo.

**Cardinalidades:**

- Detallan cuántas instancias de una clase pueden estar asociadas con otra, crucial para definir la naturaleza de las relaciones entre clases.

# Depuracion

1. **Tipos de Errores:**
    - **Errores de compilación:** detectados durante la compilación debido a errores sintácticos.
    - **Errores lógicos o "bugs":** más difíciles de detectar porque el programa se compila correctamente pero no funciona como se esperaba.

**Funciones del Depurador:**

- Herramienta integrada en IDEs como Eclipse que ayuda a resolver errores lógicos.
- Permite analizar el código en ejecución, establecer puntos de ruptura, suspender la ejecución, ejecutar paso a paso, e inspeccionar el contenido de las variables.

**Uso de Puntos de Ruptura**

- Permite detener la ejecución en puntos específicos para inspeccionar el estado del programa.
- Se pueden configurar a nivel de línea o método, y se pueden activar o desactivar según sea necesario

**Si has llegado hasta el final m debes 10 pavs**