|  |
| --- |
| **UNIVERSIDAD PRIVADA DEL NORTE** |
|  |
| **FACULTAD DE INGENIERÍA** |
| **INGENIERÍA DE SISTEMAS COMPUTACIONALES** |
|  |
|  |
| **"Consigna de Entrega”**  **Informe y Software Práctica de Campo**  **Semana 03 y 04"** |
|  |
| **ARENAS NARBASTE, DANIEL ABELARDO (N00171207)** |
|  |
| **PERÚ** |
|  |
| **2025** |

CAPITULO 1: INTRODUCCIÓN

1. Objetivo

Uso eficiente de cada uno de los parámetros indicados (arrays, la recursividad, las funciones, las clases y el manejo de archivos) para desarrollar un sistema de gestión de incidencias utilizando como base el lenguaje Java con Programación Orientada a Objetos (POO) y la librería llamada “Swing” para la interfaz gráfica, de esta manera crear un software de calidad, organizado y fácil de comprender a fin de entender la importancia de uso de los parámetros solicitados.

Uso y presentación del trabajo mediante repositorio de GitHub, comandos git add y git commit, visualizar historial de commits, manejo de branches y merges; crear y cambiar entre ramas (branches) utilizando comandos got Branch y git checkout.

1. Características del trabajo

El presente trabajo consta del desarrollo de un software de gestión de incidencias, el cual permita tener en cuenta los problemas usados al usar el software ERP de una empresa.

1. Importancia

En la mayoría de software sea pequeño, mediana o gran escala, no siempre se implementa el uso de la recursividad o excepciones; siendo las listas y clases las más usadas al momento de programar. La consecuencia de no usar la recursividad o excepciones, aunque no parezca es muy notoria y a la larga, brinda dificultad al programador o equipo de programadores, los cuales podrían ser:

* **Código espagueti**: Sin recursividad, el código puede volverse más extenso y menos mantenible, aumentando el riesgo de errores, así como también el principal problema de no ser de fácil entendimiento.
* **Fallas abruptas**: Sin un manejo adecuado de excepciones, el programa puede fallar inesperadamente, lo que resulta en una mala experiencia para el usuario.
* **Difícil depuración**: Sin excepciones, puede ser complicado identificar y corregir errores, ya que el programa puede no proporcionar información útil sobre lo que salió mal.
* **Desorganización**: Sin clases, el código puede volverse desordenado y difícil de seguir, lo que complica su mantenimiento y evolución.
* **Repetición de código**: La falta de clases puede llevar a la duplicación del código, lo que aumenta el riesgo de errores y hace que el programa sea más difícil de actualizar.

Por ello, es muy recomendable el empleo de arrays, recursividad, funciones y clases al momento de programar, no solo en lenguaje Java, sino en cualquier lenguaje de programación; ya que esto no solo afecta el orden y la calidad del código, sino también la experiencia del usuario final al momento de usarlo, o en todo caso para su fácil entendimiento para la capacidad de respuesta ante errores y cambios futuros.

Recalcando que el uso de clases y funciones en programación es fundamental para estructurar y organizar el código de manera eficiente, modular y reutilizable. Estas herramientas permiten dividir un programa grande en partes más pequeñas y manejables, facilitando su mantenimiento, ampliación y depuración a futuro.

* 1. Convenciones de nombres

Las clases deben nombrarse utilizando el formato PascalCase, donde cada palabra comienza con una letra mayúscula (ejemplo: MiClaseEjemplo). Los métodos y las variables deben utilizar el formato camelCase, donde la primera palabra comienza con minúscula y las siguientes con mayúscula (ejemplo: miMetodoEjemplo, miVariableEjemplo). Los paquetes deben estar en minúsculas y, generalmente, se utilizan nombres de dominio invertidos (ejemplo: com.ejemplo.proyecto). Estas convenciones ayudan a los desarrolladores a entender rápidamente el propósito de cada elemento en el código.

* 1. Sobre los mensajes commit

Los mensajes de commit claros son esenciales para la colaboración efectiva en un proyecto. Proporcionan contexto sobre los cambios realizados, lo que facilita la revisión del historial del proyecto y la identificación de errores. Un buen mensaje de commit debe ser breve pero descriptivo, indicando qué se cambió y por qué.

* 1. Sobre el uso de ramas

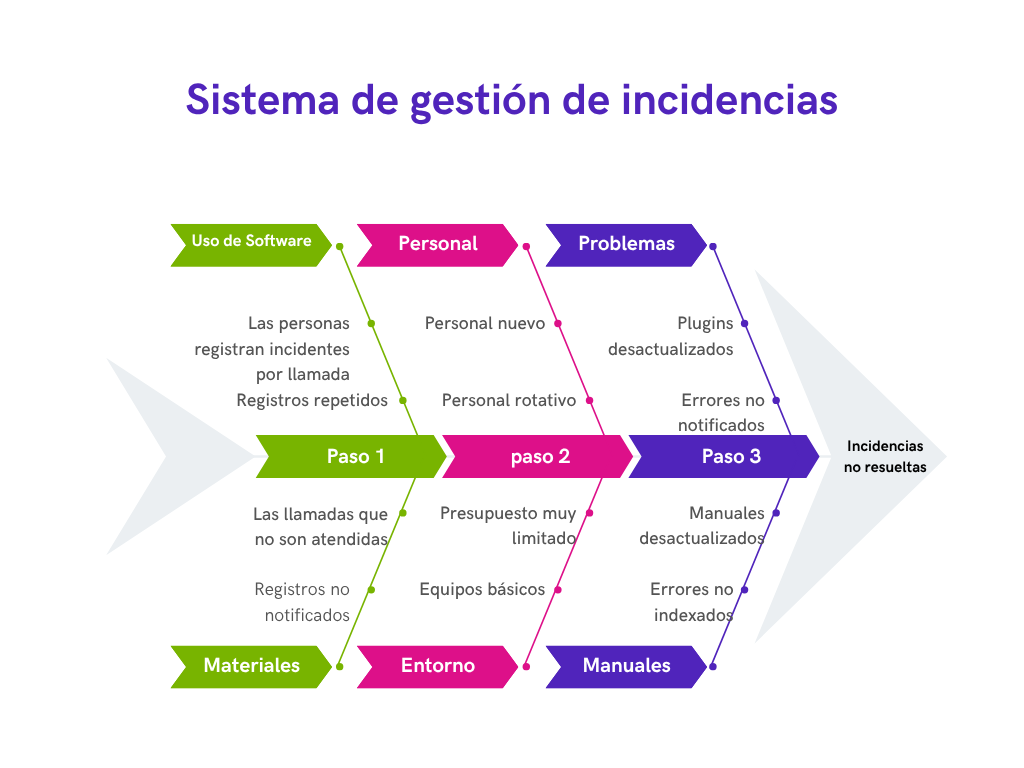
El uso de ramas en el desarrollo de software permite a los equipos trabajar en diferentes características o correcciones de errores de manera aislada. Esto minimiza el riesgo de conflictos y permite una integración más fluida de nuevas funcionalidades.

* 1. Convenciones de nombres
* **git stash**: Este comando se utiliza para guardar temporalmente los cambios no confirmados en el área de trabajo. Por ejemplo, si un desarrollador necesita cambiar de rama, pero no está listo para hacer un commit, puede usar git stash para guardar su trabajo actual.
* **git revert**: Este comando se utiliza para deshacer cambios en el historial de commits. Por ejemplo, si un commit introdujo un error, se puede revertir con git revert <commit\_id>, creando un nuevo commit que deshace los cambios del commit especificado.
* **git cherry-pick**: Este comando permite aplicar cambios de un commit específico en otra rama. Por ejemplo, si se desea aplicar un cambio de la rama feature a develop, se puede usar git cherry-pick <commit\_id> para llevar ese cambio a la nueva rama.

CAPITULO 2: FUNDAMENTOS DE SOFTWARE

1. Parámetros a usar en el software
   1. Calidad

* La sobrecarga de métodos es una característica de la programación orientada a objetos que permite definir múltiples métodos con el mismo nombre pero con diferentes parámetros. Esto es útil para realizar la misma operación con diferentes tipos o números de argumentos.
* El manejo de errores es fundamental para garantizar que un programa se ejecute de manera robusta. En Java, se utilizan bloques try-catch para capturar excepciones y manejar errores de manera controlada.
* Las colecciones en Java, como ArrayList y HashMap, son estructuras de datos que permiten almacenar y manipular grupos de objetos de manera eficiente. Su uso es fundamental para gestionar datos dinámicos y realizar operaciones de búsqueda y manipulación de manera rápida.
  1. Diagrama de Ishikawa



* 1. Requerimientos funcionales

|  |  |  |
| --- | --- | --- |
| Código | Detalle | Estado |
| RF01 | Crear un sistema de logeo con autenticación por correo y clave | Realizado |
| RF02 | Validar credenciales desde la base de datos SQLite | Realizado |
| RF03 | Redirigir al usuario al menú correspondiente según su rol | Realizado |
| RF04 | Crear menú principal por rol (Admin, Consultor, Cliente) | Realizado |
| RF05 | Mostrar nombre y rol del usuario en el menú principal | Realizado |
| RF06 | Crear panel para que el cliente registre incidencias | Realizado |
| RF07 | Registrar incidencia con estado inicial "Pendiente" | Realizado |
| RF08 | Crear panel para que el administrador asigne incidencias a consultores | Realizado |
| RF09 | Cambiar estado de la incidencia a "Asignado" al asignarla | Realizado |
| RF10 | Crear panel para que el consultor vea incidencias asignadas | Realizado |
| RF11 | Consultor responde incidencia y cambia estado a "Resuelto" | Realizado |
| RF12 | Cliente puede ver el estado de sus incidencias | Realizado |
| RF13 | Cliente puede ver la respuesta de una incidencia resuelta | Realizado |
| RF14 | Crear panel CRUD de usuarios (Administrador) | Realizado |
| RF15 | Agregar nuevo usuario desde interfaz | Realizado |
| RF16 | Editar usuario desde interfaz | Realizado |
| RF17 | Eliminar usuario lógicamente (activo = 0) | Realizado |
| RF18 | Ver lista de usuarios activos en una tabla | Realizado |
| RF19 | Agregar campo "respuesta" a la tabla incidencia | Realizado |
| RF20 | Mostrar respuesta en área separada para el cliente | Realizado |
| RF21 | Mostrar respuesta en área separada para el consultor | Realizado |
| RF22 | Empaquetar navegación con JFrame principal (`MenuPrincipal`) | Realizado |
| RF23 | Mostrar tabla de incidencias filtradas por usuario logueado | Realizado |
| RF24 | Aplicar eliminación lógica a incidencias | Realizado |

## Mejoras de consulta y visualización

|  |  |  |
| --- | --- | --- |
| Código | Detalle | Estado |
| RF25 | Buscar incidencias por título o ID | Realizado |
| RF26 | Filtrar incidencias por estado (Pendiente / Asignado / Resuelto) | Realizado |
| RF27 | Mostrar incidencias ordenadas por fecha de creación | Realizado |
| RF28 | Ver cantidad total de incidencias por estado | Realizado |
| RF29 | Mostrar solo incidencias activas (ocultar eliminadas) | Realizado |

## Seguridad y control

|  |  |  |
| --- | --- | --- |
| Código | Detalle | Estado |
| RF30 | Validar correo duplicado al registrar usuario | Realizado |
| RF31 | Encriptar contraseñas al guardar en la base de datos | Realizado |
| RF32 | Validar campos vacíos en formularios con mensajes amigables | Realizado |
| RF33 | Confirmar antes de cerrar sesión con mensaje de advertencia | Realizado |
| RF34 | Validar cambio de estado de incidencia solo por usuarios autorizados | Realizado |

## Campos adicionales

|  |  |  |
| --- | --- | --- |
| Código | Detalle | Estado |
| RF35 | Agregar campo de fecha de creación a las incidencias | Realizado |
| RF36 | Agregar campo de fecha de resolución cuando se marca como resuelta | Realizado |
| RF37 | Mostrar fecha de creación y resolución en tablas | Realizado |

## Exportación y respaldo

|  |  |  |
| --- | --- | --- |
| Código | Detalle | Estado |
| RF38 | Exportar incidencias a archivo CSV | Pendiente |
| RF39 | Exportar usuarios a archivo CSV | Pendiente |

## Roles y configuración

|  |  |  |
| --- | --- | --- |
| Código | Detalle | Estado |
| RF40 | Cambiar rol de un usuario existente desde PanelUsuario | Pendiente |

* 1. Historias de usuario
* Como cliente, quiero registrar una incidencia para reportar un problema con el sistema ERP.
* Como cliente, quiero ver el estado de mis incidencias para saber si han sido atendidas.
* Como cliente, quiero leer la respuesta del consultor para entender cómo resolver el problema.
* Como consultor, quiero ver solo las incidencias asignadas a mí para trabajar de forma ordenada.
* Como consultor, quiero responder una incidencia para ayudar al cliente a resolver su problema.
* Como administrador, quiero asignar incidencias a consultores para garantizar su atención.
* Como administrador, quiero registrar nuevos usuarios con diferentes roles para gestionar accesos.
* Como administrador, quiero editar los datos de los usuarios para mantener la información actualizada.
* Como administrador, quiero eliminar usuarios lógicamente para conservar el historial sin perder datos.
* Como administrador, quiero ver una lista de usuarios activos para tener control sobre el sistema.
* Como administrador, quiero tener acceso a todas las incidencias para poder gestionarlas si es necesario.
* Como cualquier usuario, quiero iniciar sesión con mis credenciales para acceder al sistema.
* Como administrador, quiero filtrar incidencias por estado para revisar solo las que requieren atención.
* Como consultor, quiero registrar la fecha de resolución para tener trazabilidad de mi trabajo.
* Como cliente, quiero buscar una incidencia por título para encontrarla rápidamente.
  1. Pruebas y validaciones

Se realizaron pruebas para asegurar que la gestión de palabras en el arreglo, incluyendo el registro, las listas y las condicionales, funcionaran correctamente. Los resultados confirmaron que cada funcionalidad cumple con su propósito de manera eficiente, garantizando el manejo adecuado del arreglo y la lista asociada.

CAPITULO 3: CONCLUSIONES Y RECOMENDACIONES

1. Conclusiones
   1. La utilidad de las clases

Las clases son fundamentales en la programación orientada a objetos, ya que permiten organizar y encapsular datos y comportamientos relacionados. Al definir clases, se promueve un enfoque más estructurado y comprensible en el desarrollo del software.

* 1. La utilidad de funciones

Las funciones son una piedra angular de la programación. Ayudan a crear código limpio, reutilizable, fácil de mantener y escalar. Al usarlas correctamente, los programadores pueden producir software más eficiente y confiable.

* 1. Cuando utilizar el manejo de excepciones

Implementar un manejo de excepciones adecuado permite anticipar fallos y ofrecer una experiencia de usuario más fluida al manejar errores de manera controlada.

* 1. Utilidad de la recursividad

La recursividad es una técnica poderosa que permite resolver problemas complejos de manera elegante y concisa. En este trabajo, se ha utilizado para implementar algoritmos de búsqueda y ordenación, demostrando su capacidad para simplificar el código y mejorar la legibilidad. Sin embargo, es importante ser consciente de su uso, ya que puede llevar a un mayor consumo de memoria si no se maneja adecuadamente.

* 1. Utilidad de array unidimensionales

Los arrays unidimensionales (o vectores en algunos lenguajes) son estructuras de datos fundamentales en la programación. Su utilidad se extiende a muchos aspectos del desarrollo de software, ya que permiten almacenar colecciones de datos de forma eficiente y organizada.

1. Recomendaciones

**Adoptar Buenas Prácticas de Programación:** Se recomienda seguir las buenas prácticas de programación, como la escritura de código limpio y bien comentado, para facilitar su mantenimiento y comprensión a largo plazo.

**Implementar Pruebas:** Incluir pruebas unitarias y funcionales en el ciclo de desarrollo para identificar y corregir errores de manera temprana, asegurando que el software cumpla con los requisitos establecidos.

**Fomentar la Colaboración:** Continuar promoviendo un ambiente de trabajo colaborativo donde se valore el intercambio de ideas y se aprovechen las fortalezas de cada miembro del equipo.

1. Referencias

Java YA. *Recursos para aprender JAVA en ECLIPSE*.  
 https://www.tutorialesprogramacionya.com/javaya/

NetMentor. *Recursividad en programación*. https://www.netmentor.es/entrada/Recursividad-programacion

UPN BlackBoard. *Métodos de sobrecarga, Static, Excepciones y Colecciones.*

https://upn.blackboard.com/ultra/courses/\_1602368\_1/outline/file/\_53724816\_1

LINK a repositorio en GitHub: https://github.com/arenas1802/Practicadecampo3