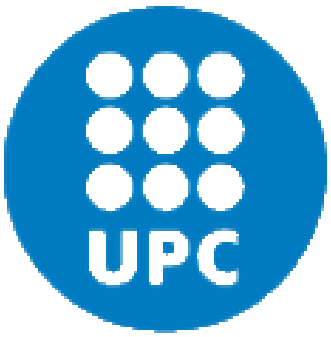


UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

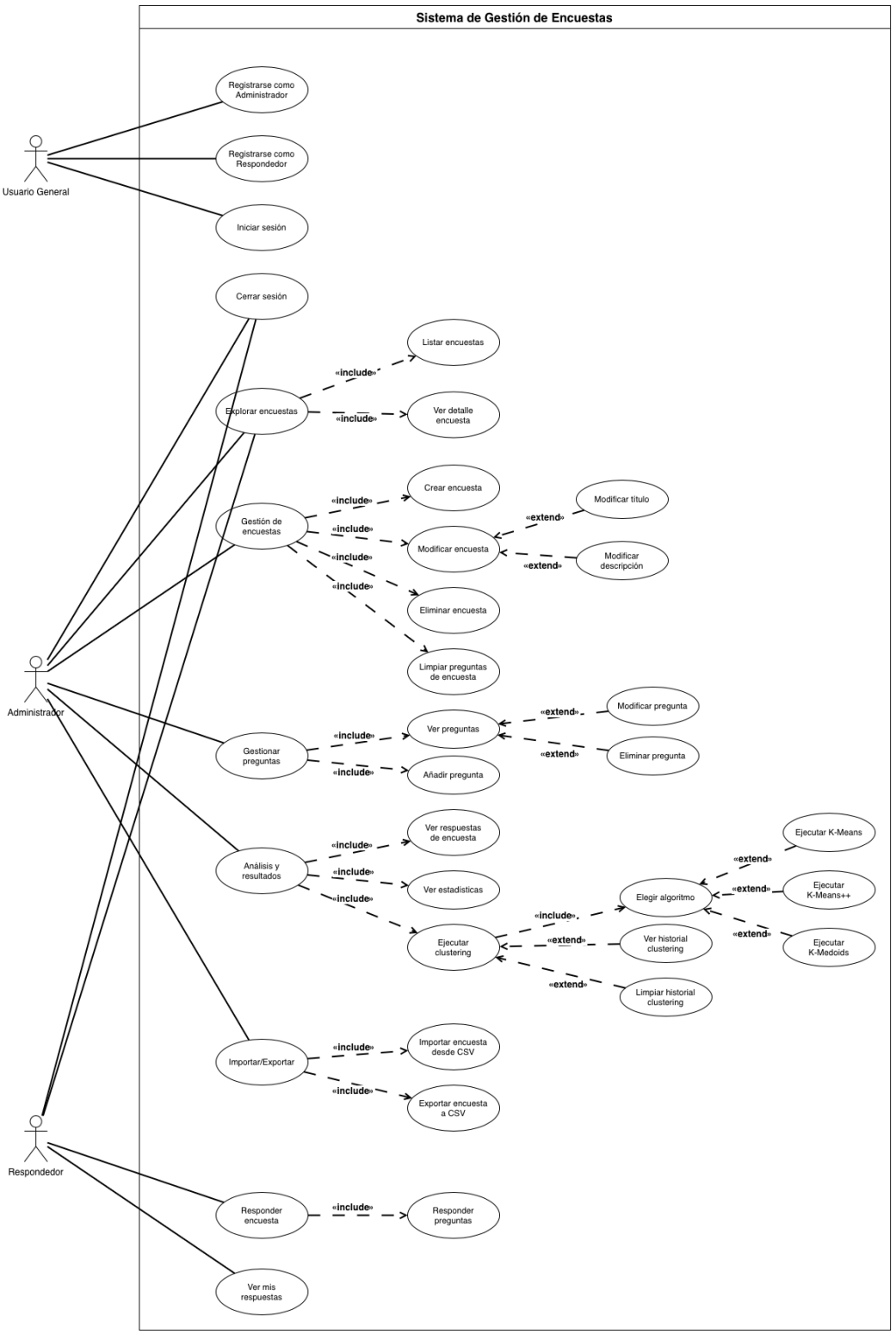
3ª ENTREGA PROP

Identificador de l'equip: 41.1

Mohamed Amara El Houti : mohamed.amara.el.houiti@estudiantat.upc.edu
Ossama Chaer Dalerou : ossama.chaer@estudiantat.upc.edu
Arnau Serra Florenciano : arnau.serra.florenciano@estudiantat.upc.edu
Adam Ziani Hassun : adam.ziani@estudiantat.upc.edu

Índice

1. Casos de Uso



1.1 Nombre: Registrarse (crear cuenta)

- **Actor:** Respondedor o Administrador.
- **Comportamiento:**
    - El usuario accede a la opción de **registrarse** e introduce un **identificador**, un **nombre visible** y una **contraseña**.
    - El sistema comprueba que los datos sean válidos y que el identificador no esté ya registrado.
    - Si todo es correcto, el sistema crea la **nueva cuenta** y confirma al usuario que el registro se ha completado.
- **Errores posibles:**
    - Si ya existe un usuario con ese identificador, el sistema informa del error y pide elegir otro.
    - Si falta algún dato obligatorio, el sistema informa de que deben completarse todos los campos.

1.2 Nombre: Iniciar sesión

- **Actor:** Respondedor o Administrador.
- **Comportamiento:**
    - El usuario introduce su identificador y contraseña.
    - El sistema verifica que el usuario existe y que la contraseña es correcta.
    - Si las credenciales son válidas, el sistema inicia la sesión y muestra el menú correspondiente al rol del usuario.
- **Errores posibles:**
    - Si el identificador no existe o la contraseña es incorrecta, el sistema muestra un mensaje de credenciales no válidas.

1.3 Nombre: Cerrar sesión

- **Actor:** Respondedor o Administrador.
- **Comportamiento:**
    - El usuario selecciona la opción de cerrar sesión.
    - El sistema finaliza la sesión actual y vuelve a la pantalla de inicio.
- **Errores posibles:**
    - No se esperan errores. Si no había sesión activa, el sistema simplemente vuelve al inicio.

1.4 Nombre: Explorar encuestas

- **Actor:** Respondedor (y también Administrador).
- **Comportamiento:**
    - El usuario selecciona la opción "*Explorar encuestas*".
    - El sistema muestra una lista con todas las encuestas disponibles.
    - El usuario puede **seleccionar** una encuesta para ver más detalles.
- **Errores posibles:**
    - Si no hay encuestas disponibles, el sistema informa de ello.
    - Si el usuario selecciona una encuesta inexistente, el sistema indica el error.

1.5 Nombre: Ver detalle de una encuesta

- **Actor:** Respondedor o Administrador.
- **Comportamiento:**
    - El usuario elige una encuesta y solicita ver sus detalles.
    - El sistema muestra el **título**, la **descripción** y la **lista de preguntas**.
- **Errores posibles:**
    - Si la encuesta ya no existe, el sistema informa del error.

1.6 Nombre: Responder preguntas

- **Actor:** Respondedor.
- **Comportamiento:**
    - El usuario inicia el **proceso de respuesta** de una encuesta.
    - Para cada pregunta, el sistema muestra su enunciado y el tipo de respuesta esperado.
    - El usuario introduce su respuesta y el sistema la valida.
    - Tras completar todas las preguntas, el sistema confirma que la encuesta ha sido respondida correctamente.
- **Errores posibles:**
    - Si una pregunta obligatoria queda sin respuesta, el sistema pide completarla.
    - Si la respuesta no tiene el formato esperado, el sistema pide corregirla.
    - Si se excede el número permitido de opciones en preguntas de selección múltiple, el sistema lo indica.
    - Si el usuario abandona la encuesta, el sistema informa que la encuesta queda incompleta.

1.7 Nombre: Crear encuesta

- **Actor:** Administrador.
- **Comportamiento:**
    - El administrador elige la opción "*Crear nueva encuesta*".
    - El sistema solicita el título y la descripción.
    - Al confirmar, el sistema crea una encuesta **vacía**.
- **Errores posibles:**
    - Si el usuario no es administrador, el sistema deniega la acción.
    - Si el título está vacío, el sistema indica que debe incluirse un título válido.

1.8 Nombre: Importar encuesta (desde CSV)

- **Actor:** Administrador.
- **Comportamiento:**
    - El administrador selecciona la opción de **importar** y proporciona la ruta del archivo **CSV**.
    - El sistema valida que el archivo existe y tiene el **formato adecuado**.
    - Muestra una vista previa con número de preguntas, respuestas y tipos detectados.
    - Si se confirma, el sistema crea la encuesta, genera las preguntas y carga los usuarios y sus respuestas.
- **Errores posibles:**
    - Archivo inexistente o ilegible.
    - Falta de cabeceras o filas de datos.
    - Tipos de preguntas no válidos.
    - Columnas numéricas con valores no numéricos.
    - Preguntas categóricas con menos de dos opciones.
    - El administrador puede cancelar antes de confirmar.

1.9 Nombre: Listar encuestas

- **Actor:** Administrador.
- **Comportamiento:**
    - El administrador solicita ver todas las encuestas.
    - El sistema muestra la lista completa.
- **Errores posibles:**
    - Si no hay encuestas, el sistema indica que la lista está vacía.

1.10 Nombre: Añadir pregunta

- **Actor:** Administrador.
- **Comportamiento:**
    - El administrador selecciona la encuesta y elige **añadir** pregunta.
    - El sistema solicita el enunciado y el tipo de pregunta.
    - Según el tipo, pide configuraciones adicionales (opciones, rangos, etc.).
    - Al confirmar, el sistema añade la pregunta a la encuesta.
- **Errores posibles:**
    - Si la encuesta no existe, el sistema informa del error.
    - Si se introducen opciones insuficientes para preguntas categóricas, el sistema pide corregirlo.
    - Si se define un rango numérico inválido, el sistema avisa.

1.11 Nombre: Eliminar pregunta

- **Actor:** Administrador.
- **Comportamiento:**
    - El administrador selecciona una pregunta y solicita **eliminarla**.
    - El sistema pide confirmación y, si se aprueba, la borra.
- **Errores posibles:**
    - Si el índice de la pregunta no existe, el sistema avisa.
    - Si la encuesta no existe, se informa del error.

1.12 Nombre: Limpiar encuesta

- **Actor:** Administrador.
- **Comportamiento:**
    - El administrador selecciona "*Eliminar todas las preguntas*".
    - El sistema pide confirmación y, si se acepta, **elimina todas las preguntas**.
- **Errores posibles:**
    - Si la encuesta no existe, el sistema no realiza la acción.
    - Si se cancela la confirmación, no se hace ningún cambio.

1.13 Nombre: Modificar pregunta

- **Actor:** Administrador.
- **Comportamiento:**
    - El administrador selecciona la pregunta a **modificar**.
    - El sistema muestra la configuración **actual**.
    - El administrador introduce los cambios y el sistema actualiza la pregunta.
- **Errores posibles:**
    - Si la pregunta no existe, el sistema indica el error.
    - Si los datos nuevos son inválidos, el sistema solicita corregirlos.

1.14 Nombre: Eliminar encuesta

- **Actor:** Administrador.
- **Comportamiento:**
    - El administrador selecciona una encuesta y pide **eliminarla**.
    - El sistema pide confirmación.
    - Si se confirma, elimina la encuesta y sus datos.
- **Errores posibles:**
    - Si la encuesta no existe, el sistema informa del error.

1.15 Nombre: Exportar encuesta a CSV

- **Actor:** Administrador.
- **Comportamiento:**
    - El administrador selecciona una encuesta y elige **exportarla**.
    - El sistema comprueba que tiene preguntas y respuestas.
    - El sistema pide un nombre de archivo y genera el **CSV**.
- **Errores posibles:**
    - Encuesta sin preguntas o sin respuestas.
    - Problemas al escribir el archivo (permisos, ruta inválida).

1.16 Nombre: Ver todas las respuestas

- **Actor:** Administrador.
- **Comportamiento:**
    - El administrador **selecciona** una encuesta.
    - El sistema reúne las respuestas de todos los usuarios y las muestra en una tabla.

- **Errores posibles:**
    - Encuesta inexistente.
    - No hay respuestas disponibles.

1.17 Nombre: Ver clusterings

- **Actor:** Administrador.
- **Comportamiento:**
    - El administrador selecciona una encuesta y elige la opción "*Ver clusterings*".
    - El sistema localiza los análisis de clustering **previamente ejecutados** sobre esa encuesta.
    - El sistema muestra una lista con todos los clusterings disponibles, indicando para cada uno:
        - el algoritmo utilizado (*K-Means, K-Means++, K-Medoids*).
        - el número de grupos k.
        - y el valor del índice de calidad (por ejemplo, silhouette).
    - El administrador puede seleccionar uno de los clusterings para visualizar sus detalles (si el sistema lo permite en otros casos de uso).
- **Errores posibles:**
    - Si la encuesta no existe, el sistema informa de que no se pueden recuperar los clusterings.
    - Si aún no se ha ejecutado ningún clustering para esa encuesta, el sistema informa de que no hay análisis disponibles.
    - Si se selecciona un clustering inexistente, el sistema indica el error y solicita elegir uno válido.

1.18 Nombre: Ejecutar clustering

- **Actor:** Administrador.
- **Comportamiento:**
    - El administrador elige una encuesta y selecciona la opción de *análisis por clustering*.
    - El sistema solicita el algoritmo (*K-Means, K-Means++, K-Medoids*) y el número de grupos.
    - El sistema ejecuta el análisis y muestra los grupos encontrados y el valor del indicador de calidad.
- **Errores posibles:**
    - Valor de k inválido.
    - Encuesta sin respuestas completas.
    - Algoritmo no reconocido.

1.19 Nombre: Ejecutar K-Means / Ejecutar K-Means++ / Ejecutar K-Medoids

- **Actor:** Administrador.

- **Comportamiento:**

    - El administrador elige la variante deseada del clustering.
    - El sistema ejecuta el análisis y muestra los grupos obtenidos.

- **Errores posibles:**

    - Valor de k inválido.
    - Encuesta sin respuestas completas.
    - Algoritmo no reconocido.

1.20 Nombre: Ver mis respuestas

- **Actor:** Respondedor
- **Comportamiento:**
    - El usuario respondedor selecciona la opción "*Ver mis respuestas*"
    - El sistema solicita el **ID** de la encuesta
    - El sistema muestra todas las respuestas que el usuario ha dado a esa encuesta específica
    - Para cada pregunta se muestra el enunciado y la respuesta proporcionada
- **Errores posibles:**
    - Si la encuesta no existe, el sistema informa del error
    - Si el usuario no ha respondido esa encuesta, el sistema informa que no hay respuestas

1.21 Nombre: Ver estadísticas de encuesta

- **Actor:** Administrador
- **Comportamiento:**
    - El administrador selecciona una encuesta
    - El sistema muestra estadísticas generales: título, número de preguntas, número de respuestas totales
    - El sistema lista los usuarios que han respondido la encuesta
- **Errores posibles:**
    - Si la encuesta no existe, el sistema informa del error

1.22 Nombre: Ver historial de clustering

- **Actor:** Administrador
- **Comportamiento:**
    - El administrador selecciona una encuesta y elige "*Ver historial de clustering*"
    - El sistema muestra todas las ejecuciones previas de clustering realizadas sobre esa encuesta
    - Para cada ejecución se muestra: algoritmo, K, Silhouette, Inercia
    - El administrador puede seleccionar una ejecución **específica** para ver sus detalles completos

- **Errores posibles:**
  - Si la encuesta no existe, el sistema informa del error
  - Si no hay ejecuciones previas, el sistema informa que el historial está vacío
  - Si se selecciona un índice inválido, el sistema indica el error

1.23 Nombre: Limpiar historial de clustering

- **Actor:** Administrador
- **Comportamiento:**
  - El administrador selecciona una encuesta y elige "*Limpiar historial de clustering*"
  - El sistema muestra cuántos resultados se van a **eliminar**
  - El sistema pide **confirmación explícita**
  - Si se confirma, el sistema **elimina todos los resultados de clustering** de esa encuesta
- **Errores posibles:**
  - Si la encuesta no existe, el sistema informa del error
  - Si no hay historial, el sistema informa que no hay nada que limpiar
  - Si se cancela la confirmación, no se realiza ningún cambio

1.24 Nombre: Ver centros de clusters

- **Actor:** Administrador
- **Comportamiento:**
  - Tras visualizar un resultado de clustering, el administrador puede solicitar ver los centros
  - El sistema muestra los valores centrales de cada cluster para cada pregunta
  - Se presenta de forma estructurada: Centro 1, Centro 2, etc., con los valores por pregunta
- **Errores posibles:**
  - Si el resultado de clustering no tiene centros calculados, el sistema informa del error

1.25 Nombre: Ver tabla comparativa de K's

- **Actor:** Administrador
- **Comportamiento:**
  - Al ejecutar clustering con múltiples valores de K, el sistema automáticamente genera una **tabla comparativa**
  - La tabla muestra para cada K: Silhouette Score, Inercia, y número de iteraciones
  - El sistema marca automáticamente el **K óptimo** (mayor Silhouette)
- **Errores posibles:**
  - Si no se pudieron generar resultados para ningún K, el sistema informa del error

1.26 Nombre: Modificar encuesta (título y descripción conjuntamente)

- **Actor:** Administrador
- **Comportamiento:**
  - El administrador selecciona "*modificar encuesta*"
  - El sistema permite cambiar **título** y **descripción** en una sola operación
  - Se puede dejar algún campo vacío para **mantener el valor actual**
  - El sistema actualiza solo los campos que tienen nuevo valor
- **Errores posibles:**
  - Si la encuesta no existe, el sistema informa del error
  - Si ambos campos están vacíos, no se realiza ningún cambio

1.27 Nombre: Ver preguntas detallado

- **Actor:** Administrador
- **Comportamiento:**
  - El administrador selecciona una encuesta
  - El sistema muestra todas las preguntas con información completa:
    - Número de orden
    - Enunciado
    - Tipo de pregunta
    - Si es obligatoria
    - Opciones/rangos/configuraciones específicas según el tipo

- **Errores posibles:**
  - Si la encuesta no existe, el sistema informa del error
  - Si no hay preguntas, el sistema informa que la encuesta está vacía

2. Diagramas de las capas y su descripción

2.1. Capa de dominio

2.1.1. ControladorEncuestas

Responsabilidad: Gestión de la lógica de negocio relacionada con encuestas. Opera únicamente con objetos en memoria, sin interactuar directamente con la capa de persistencia.

Atributos: Ninguno (clase sin estado, sin dependencias de persistencia).

Métodos principales:

- ***crearEncuesta(titulo, descripcion):*** Crea una nueva instancia de Encuesta. Valida que el título no esté vacío ni sea nulo. Retorna el objeto Encuesta creado con un ID único generado.
- ***modificarEncuesta(encuesta, nuevoTitulo, nuevaDescripcion)***: Actualiza los datos básicos de una encuesta existente. Permite modificar título y/o descripción. Valida que la encuesta no sea nula. Aplica los cambios mediante los setters correspondientes.
- ***agregarPregunta(encuesta, pregunta)****:* Añade una nueva pregunta al final de la lista de preguntas de la encuesta. Valida que tanto la encuesta como la pregunta no sean nulas. Delega la operación al método agregarPregunta() de la clase Encuesta.
- ***modificarPregunta(encuesta, indice, nuevaPregunta)****:* Reemplaza una pregunta existente en una posición específica. Valida el índice (debe estar dentro del rango válido). Lanza IndexOutOfBoundsException si el índice es inválido. Utiliza el método modificarPregunta() de Encuesta.
- ***eliminarPregunta(encuesta, indice)****:* Elimina la pregunta ubicada en el índice especificado. Valida que el índice sea válido antes de eliminar. Delega la eliminación al método eliminarPregunta() de Encuesta.
- ***validarEncuesta(encuesta)****:* Verifica la integridad básica de una encuesta. Comprueba que la encuesta no sea nula y que tenga un título válido (no nulo ni vacío). Retorna booleano indicando si la encuesta es válida.
- ***buscarPregunta(encuesta, idPregunta)****:* Localiza una pregunta dentro de la encuesta mediante su identificador único. Itera sobre la lista de preguntas comparando IDs. Retorna la pregunta encontrada o null si no existe.
- ***obtenerIndicePregunta(encuesta, idPregunta)****:* Determina la posición numérica de una pregunta en la lista. Retorna el índice (0-based) o -1 si la pregunta no se encuentra en la encuesta.

2.1.2. ControladorUsuarios

Responsabilidad: Gestión centralizada de usuarios del sistema, tanto administradores como respondedores. Maneja la creación y validación de usuarios.

Atributos: Ninguno (clase sin estado).

Métodos principales:

- ***crearUsuarioRespondedor(id, nombre, password)****:* Crea una nueva instancia de UsuarioRespondedor con autenticación. Valida que ID y nombre no estén vacíos mediante validarDatosUsuario(). Retorna el objeto UsuarioRespondedor creado.
- ***crearUsuarioRespondedor(id, nombre)****:* Sobrecarga del método anterior para crear respondedor sin contraseña. Útil para importaciones CSV donde no se requiere autenticación.
- ***crearUsuarioAdmin(id, nombre, password)****:* Crea una nueva instancia de UsuarioAdmin con privilegios administrativos. Valida datos mediante validarDatosUsuario(). Retorna el objeto UsuarioAdmin creado.
- ***filtrarUsuariosQueRespondieron(usuarios, idEncuesta)****:* Filtra una lista de usuarios retornando únicamente aquellos que han completado al menos una respuesta en la encuesta especificada. Utiliza el método haRespondidoEncuesta() de cada usuario. Retorna nueva lista con usuarios filtrados.
- ***obtenerRespuestasUsuario(usuario, idEncuesta)****:* Extrae todas las respuestas de un usuario específico para una encuesta dada. Retorna lista vacía si el usuario es nulo. Delega la operación a getRespuestasEncuesta() del usuario.
- ***validarDatosUsuario(id, nombre)****:* Método privado que verifica la validez de los datos básicos de un usuario. Comprueba que ID y nombre no sean nulos ni estén vacíos (trim). Lanza IllegalArgumentException con mensaje descriptivo si alguna validación falla.
- ***esUsuarioValido(usuario)****:* Validación completa de un objeto Usuario. Verifica que el usuario no sea nulo y que sus atributos ID y nombre sean válidos. Retorna booleano indicando validez completa.

2.1.2. ControladorRespuestas

Responsabilidad: Gestión del proceso de responder encuestas. Maneja la creación, validación y recuperación de respuestas de usuarios.

Atributos: Ninguno (clase sin estado).

Métodos principales:

- ***responderEncuesta(usuario, encuesta, respuestas)****:* Registra el conjunto completo de respuestas de un usuario a una encuesta. Recibe Map<String, Object> donde la clave es idPregunta y el valor es la respuesta. Para cada respuesta: busca la pregunta correspondiente, valida el valor contra el tipo de pregunta, crea objeto

Respuesta y lo añade al usuario. Retorna Map<String, Respuesta> con las respuestas creadas. Lanza excepciones si alguna pregunta no existe o si algún valor no es válido.

- ***responderPregunta(usuario, encuesta, idPregunta, valor)****:* Registra una respuesta individual a una pregunta específica. Busca la pregunta en la encuesta, valida el valor mediante validarRespuesta(), crea objeto Respuesta con todos los identificadores necesarios (idUsuario, idPregunta, idEncuesta), añade la respuesta al usuario mediante addRespuesta(). Retorna el objeto Respuesta creado.
- ***obtenerRespuestas(usuario, idEncuesta)****:* Recupera todas las respuestas que un usuario ha dado a una encuesta. Retorna lista vacía si el usuario es nulo. Delega a getRespuestasEncuesta() del usuario.
- ***validarRespuestas(encuesta, respuestas)****:* Valida un conjunto completo de respuestas antes de persistirlas. Para cada entrada en el mapa: busca la pregunta correspondiente, verifica que la pregunta exista en la encuesta, valida el valor contra el tipo de pregunta. Retorna false si alguna validación falla. Útil para validación preventiva antes de guardar.
- ***buscarPregunta(encuesta, idPregunta)****:* Método privado auxiliar que localiza una pregunta dentro de una encuesta. Itera sobre la lista de preguntas comparando IDs. Retorna la Pregunta o null si no existe.

2.1.4. CtrlClustering

Responsabilidad: Coordinación y ejecución de algoritmos de clustering sobre datos de encuestas. Gestiona múltiples ejecuciones y selección del mejor resultado.

Atributos:

- ***clustering***: Instancia de la clase Clustering que realiza la preparación de datos y ejecución efectiva de algoritmos.

Métodos principales:

- ***ejecutarClustering(encuesta, usuarios, indicePregunta, algoritmo, kMax, maxIter)****:* Ejecuta clustering para múltiples valores de K (desde 2 hasta kMax). Para cada valor de K: ejecuta el algoritmo 10 veces con diferentes semillas, selecciona el mejor resultado (mayor Silhouette), añade resultado a la lista. Valida parámetros antes de iniciar. Ajusta kMax al número de usuarios si es necesario (no puede haber más clusters que usuarios). Retorna List<ResultadoClustering> con el mejor resultado para cada K.
- ***ejecutarParaK(k, algoritmo, maxIter, usuarios, encuesta, indicePregunta)****:* Método privado que ejecuta clustering múltiples veces (10 ejecuciones) para un K específico. Para cada ejecución: crea nueva estrategia (algoritmo), configura semilla única (ejecucion * 1000L) para reproducibilidad, ejecuta clustering mediante la clase Clustering, compara Silhouette con el mejor actual. Retorna el ResultadoClustering con mayor Silhouette de las 10 ejecuciones.
- ***encontrarMejorResultado(resultados)****:* Analiza una lista de resultados de clustering y selecciona el óptimo global. Compara coeficientes de Silhouette (valores entre -1 y 1, mayor es mejor). Retorna el ResultadoClustering con mayor Silhouette, o null si la lista está vacía.
- ***validarParametros(kMax, encuesta, usuarios)****:* Método privado que verifica precondiciones antes de ejecutar clustering. Comprueba: kMax >= 2 (mínimo dos clusters), encuesta no nula, lista de usuarios no nula ni vacía. Lanza IllegalArgumentException o IllegalStateException con mensajes descriptivos.
- ***crearEstrategia(algoritmo, k, maxIter)****:* Factory method que instancia el algoritmo de clustering apropiado. Acepta strings: "KMEANS"/"1", "KMEANS++"/"2", "KMEDOIDS"/"3". Crea instancia con parámetros k (número de clusters) y maxIter (iteraciones máximas). Lanza IllegalArgumentException si el algoritmo no es reconocido.

2.1.5. ControladorImportacion

Responsabilidad: Transformación de datos CSV crudos en objetos de dominio completos. Realiza inferencia de tipos de preguntas y creación automática de la estructura de encuesta.

Atributos:

- ***controladorUsuarios****:* Dependencia para crear objetos Usuario
- ***controladorEncuestas***: Dependencia para crear Encuesta y Preguntas
- ***controladorRespuestas***: Dependencia para registrar respuestas

Métodos principales:

- ***procesarDatosCSV(datosCSV, titulo, descripcion)****:* Orquesta todo el proceso de importación. Extrae encabezados (nombres de preguntas) ignorando primera columna (ID usuario). Crea encuesta vacía con título y descripción. Infiere tipos de cada columna analizando los datos. Crea preguntas según tipos inferidos. Procesa todas las filas creando usuarios y sus respuestas. Retorna ResultadoImportacion con encuesta, usuarios y mapa de respuestas. Imprime mensajes de progreso en consola.

- ***inferirTiposDeColumnas(filas, numColumnas)****:* Analiza el contenido de cada columna para determinar automáticamente el tipo de pregunta más apropiado. Para cada columna: intenta parsear todos los valores como Double (si todos son numéricos → NUMERICA o CATEGORIA_SIMPLE si hay ≤2 valores únicos), cuenta valores únicos (si ≤10 valores únicos → CATEGORIA_SIMPLE, si >10 → LIBRE), calcula min/max para numéricas con margen del 5%. Retorna List<TipoInferido> con tipo y metadata (opciones, rangos).
- ***crearPreguntasDesdeInferencia(encuesta, nombres, tiposInferidos)****:* Crea objetos Pregunta concretos según los tipos inferidos. Para cada columna: crea instancia de Numerica (con min/max), CategoriaSimple (con opciones), o Libre (longitud 1000). Añade cada pregunta a la encuesta mediante agregarPregunta().
- ***procesarRespuestas(encuesta, filasRespuestas)****:* Procesa cada fila del CSV creando usuarios y registrando sus respuestas. Para cada fila: extrae ID de usuario, crea UsuarioRespondedor, asigna respuestas llamando a asignarRespuestasUsuario(), maneja errores por fila sin detener el proceso completo. Retorna ResultadoImportacion con usuarios creados y sus respuestas.
- ***asignarRespuestasUsuario(usuario, encuesta, respuestas)****:* Método privado que asigna las respuestas de un usuario. Para cada posición en el array de respuestas: obtiene la pregunta correspondiente, convierte el valor String al tipo apropiado, registra respuesta mediante controladorRespuestas. Maneja valores vacíos asignándolos como null. Captura excepciones por pregunta individual.
- ***convertirValor(valor, pregunta)****:* Convierte un String del CSV al tipo de dato apropiado según el tipo de pregunta. Para NUMERICA: parsea como Double. Para ORDINAL/CATEGORIA_SIMPLE: mantiene como String. Para CATEGORIA_MULTIPLE: divide por comas y crea Set<String>. Para LIBRE: mantiene como String. Lanza IllegalArgumentException con mensaje descriptivo si la conversión falla.

2.1.6. ControladorExportacion

Responsabilidad: Preparación de datos de dominio para exportación a formato CSV. No escribe archivos, solo prepara la estructura de datos.

Atributos: Ninguno (clase sin estado).

Métodos principales:

- ***prepararExportacion(encuesta, usuarios, respuestasPorUsuario)****:* Orquesta el proceso completo de preparación. Valida que encuesta tenga preguntas y usuarios no esté vacío. Crea encabezados mediante crearEncabezados(). Crea filas de datos mediante crearFilas(). Retorna DatosExportacion con estructura lista para escribir. Lanza IllegalArgumentException si las validaciones fallan.
- ***obtenerInfoExportacion(encuesta, usuarios)****:* Genera metadata sobre la exportación sin ejecutarla. Cuenta número de preguntas y usuarios. Retorna InfoExportacion con estadísticas (útil para mostrar al usuario antes de confirmar exportación).

- ***crearEncabezados(preguntas)****:* Método privado que construye la primera fila del CSV. Crea array con tamaño = preguntas + 1. Primera posición: "ID_Usuario". Posiciones siguientes: enunciado de cada pregunta. Retorna String[] con encabezados.
- ***crearFilas(preguntas, usuarios, respuestasPorUsuario)****:* Método privado que construye la matriz de datos. Para cada usuario: crea fila con ID en primera posición, mapea respuestas por ID de pregunta para acceso rápido, llena posiciones según orden de preguntas (usa formatearRespuesta() para conversión), asigna "" si no hay respuesta. Retorna List<String[]> con todas las filas.
- ***formatearRespuesta(valor, pregunta)****:* Método privado que convierte objetos Java a strings apropiados para CSV. Para null: retorna "". Para NUMERICA con Double: si es entero lo muestra sin decimales, sino con decimales. Para CATEGORIA_MULTIPLE con Set: une elementos con comas. Para otros tipos: usa toString(). Preserva formato que permite re-importación.

Clase interna *DatosExportacion*:

- **Atributos**: encabezados (String[]), filas (List<String[]>)
- **Método toList()**: Combina encabezados y filas en una sola lista para facilitar escritura
- **Uso**: Empaqueta datos preparados para que la capa de persistencia los escriba

Clase interna *InfoExportacion*:

- **Atributos**: *numeroPreguntas (int), numeroUsuarios (int), tituloEncuesta (String)*
- **Método *toString()****:* Formatea información legible para usuario
- **Uso**: Preview de la exportación antes de ejecutarla

2.1.7. CtrlDominio

Responsabilidad: Fachada principal que coordina todos los controladores de dominio y gestiona la comunicación con la capa de persistencia. Punto de entrada único para la capa de presentación.

Atributos:

- ***persistencia****:* Instancia de CtrlPersistencia para operaciones de almacenamiento
- ***gson****:* Instancia de Gson con configuración para pretty printing
- ***controladorEncuestas, controladorUsuarios, controladorRespuestas****:* Controladores de lógica de negocio
- ***controladorClustering, controladorImportacion, controladorExportacion****:* Controladores especializados
- ***adaptadorPregunta****:* Adaptador para conectar Clustering con índices de preguntas
- ***persistenciaCSV****:* Acceso directo a operaciones CSV

Métodos de Encuestas:

- ***crearEncuesta(titulo, descripcion)****:* Crea encuesta mediante controladorEncuestas. Serializa a JSON usando encuestaToJson(). Persiste mediante persistencia.guardarEncuesta(). Retorna objeto Encuesta creado.
- ***obtenerEncuesta(idEncuesta)****:* Recupera JSON de persistencia. Deserializa usando *jsonToEncuesta()*. Retorna Encuesta o null si no existe.
- ***listarEncuestas()****:* Obtiene todos los JSON de encuestas. Deserializa cada uno. Retorna List<Encuesta>.
- ***modificarEncuesta(idEncuesta, titulo, descripcion)****:* Obtiene encuesta, modifica mediante controlador, serializa y persiste. Retorna boolean indicando éxito.
- ***eliminarEncuesta(idEncuesta)****:* Elimina encuesta, sus respuestas y resultados de clustering asociados. Retorna boolean indicando éxito.

Métodos de Preguntas:

- ***addPregunta(idEncuesta, pregunta)****:* Obtiene encuesta, añade pregunta mediante controlador, serializa y persiste encuesta actualizada.
- ***modificarPregunta(idEncuesta, index, pregunta)****:* Obtiene encuesta, modifica pregunta en índice, serializa y persiste.
- ***eliminarPregunta(idEncuesta, index)****:* Obtiene encuesta, elimina pregunta, serializa y persiste.

Métodos de Usuarios:

- ***crearUsuarioRespondedor(id, nombre, password)****:* Valida que no exista usuario con ese ID. Crea mediante controlador. Serializa y persiste JSON.
- ***crearUsuarioAdmin(id, nombre, password)****:* Similar a respondedor pero crea admin y persiste en archivo separado.
- ***obtenerUsuario(idUsuario)****:* Recupera JSON, determina tipo (admin/respondedor), deserializa al tipo apropiado.
- ***listarUsuarios()****:* Lista todos los admins y respondedores, deserializa y combina en lista única.
- ***listarRespondedores()****:* Lista solo respondedores deserializados.

Métodos de Respuestas:

- ***responderEncuesta(idUsuario, idEncuesta, respuestas)****:* Obtiene usuario y encuesta. Registra respuestas mediante controlador. Serializa valores con serializarValor(). Persiste respuestas y usuario actualizado.
- ***responderPregunta(idUsuario, idEncuesta, idPregunta, valor)****:* Similar a responderEncuesta pero para una sola pregunta. Recupera respuestas existentes y añade/actualiza la nueva.
- ***obtenerRespuestasUsuario(idUsuario, idEncuesta)****:* Recupera respuestas de persistencia. Deserializa valores con deserializarValor(). Crea objetos Respuesta. Retorna List<Respuesta>.
- ***obtenerUsuariosQueRespondieron(idEncuesta)****:* Obtiene IDs de persistencia. Carga cada usuario. Retorna *List<UsuarioRespondedor>*.

Métodos de Clustering:

- ***ejecutarClustering(idEncuesta, algoritmo, kMax, maxIter)****:* Obtiene encuesta y usuarios. Ejecuta mediante controladorClustering. Serializa resultados a JSON. Persiste JSON completo. Añade mejor resultado al historial de encuesta. Retorna List<ResultadoClustering>.
- ***obtenerHistorialClustering(idEncuesta)****:* Recupera JSON de clustering. Deserializa con TypeToken para *List<ResultadoClustering*>. Retorna lista o vacía.
- ***obtenerMejorResultadoGuardado(idEncuesta)****:* Obtiene historial y selecciona mejor por Silhouette.
- ***limpiarHistorialClustering(idEncuesta)****:* Elimina resultados de persistencia y limpia historial de encuesta.

Métodos CSV:

- ***exportarEncuesta(idEncuesta, rutaArchivo)****:* Obtiene encuesta y usuarios. Recupera respuestas de cada usuario. Prepara datos mediante controladorExportacion. Escribe CSV mediante persistenciaCSV.
- ***importarCSV(rutaArchivo, titulo, descripcion)****:* Lee CSV mediante persistenciaCSV. Procesa mediante controladorImportacion (crea encuesta, usuarios, respuestas). Persiste todos los objetos creados. Retorna Encuesta importada.

Métodos de Serialización:

- ***encuestaToJson(encuesta)****:* Método privado que serializa Encuesta a JSON manejando polimorfismo de preguntas. Crea JsonObject con datos básicos. Para cada pregunta: serializa tipo y data como objeto anidado. Serializa historial de clustering. Retorna String JSON.
- ***jsonToEncuesta(json)****:* Método privado que deserializa JSON a Encuesta. Parsea JSON, crea Encuesta con datos básicos. Deserializa cada pregunta según su tipo. Restaura historial de clustering. Retorna Encuesta completa.
- ***deserializarPregunta(tipo, data)****:* Método privado que crea instancia concreta de Pregunta según TipoPregunta. Usa gson.fromJson() con clase específica (Numerica, CategoriaSimple, etc.). Retorna Pregunta o null.
- ***serializarValor(valor)****:* Convierte Object a String para persistencia. Set → "SET:" + elementos separados por "|||". Otros → toString().
- ***deserializarValor(str)****:* Convierte String de persistencia a Object. Si empieza con "SET:" → crea HashSet. Si es número → Double. Sino → String.

Métodos de Sistema:

- ***cargarTodoInmediatamente()****:* Fuerza carga de todos los datos de persistencia (encuestas, usuarios, respuestas).
- ***cerrarSistema()****:* Persiste todos los datos pendientes mediante *persistencia.guardarTodo()*.

Métodos de Consulta:

- ***esAdmin(id)****:* Delega a persistencia para verificar tipo.
- ***existeUsuario(id)****:* Delega a persistencia.
- ***existeEncuesta(id)****:* Delega a persistencia.
- ***obtenerIndicePregunta(idEncuesta, idPregunta)****:* Obtiene encuesta y delega a *controladorEncuestas*.

2.1.8. Usuario (abstracta)

Responsabilidad: Clase base para todos los usuarios del sistema. Define atributos y comportamiento común.

Atributos:

- **id**: Identificador único del usuario (String)
- **nombre**: Nombre completo o alias del usuario (String)
- **password**: Contraseña para autenticación (String, puede ser nulo)

Métodos:

- Getters y setters para todos los atributos
- Dos constructores: con password y sin password

2.1.9. UsuarioRespondedor

Responsabilidad: Usuario que puede responder encuestas. Almacena todas sus respuestas organizadas por encuesta.

Hereda de: Usuario

Atributos:

- ***respuestasPorEncuesta****: Map<String, List<Respuesta>>* que mapea idEncuesta a lista de respuestas

Métodos principales:

- ***addRespuesta(idEncuesta, respuesta)****:* Añade una respuesta a una encuesta específica. Usa computeIfAbsent para crear lista si es primera respuesta a esa encuesta. Añade respuesta a la lista correspondiente.
- ***getRespuestasEncuesta(idEncuesta)****:* Recupera todas las respuestas dadas a una encuesta. Retorna nueva ArrayList (copia defensiva) o lista vacía si no hay respuestas.
- ***haRespondidoEncuesta(idEncuesta)****:* Verifica si el usuario ha respondido al menos una pregunta de la encuesta. Comprueba existencia de clave y que lista no esté vacía.

- ***getNumeroEncuestasRespondidas()****:* Cuenta encuestas con al menos una respuesta. Itera sobre valores del mapa verificando listas no vacías.
- ***eliminarRespuestasEncuesta(idEncuesta)****:* Elimina todas las respuestas asociadas a una encuesta.

2.1.10. UsuarioAdmin

Responsabilidad: Usuario con privilegios administrativos. Puede crear y gestionar encuestas.

Hereda de: Usuario

Funcionalidad: En la versión actual no tiene lógica adicional más allá de la distinción de tipo. Su existencia permite aplicar control de acceso en la capa de presentación.

2.1.11. Encuesta

Responsabilidad: Representa una encuesta con su conjunto de preguntas y resultados de análisis.

Atributos:

- ***id****:* Identificador único generado con UUID (String)
- ***titulo****:* Título descriptivo de la encuesta (String)
- ***descripcion****:* Descripción extendida opcional (String)
- ***preguntas****: List<Pregunta>* ordenada de preguntas
- ***historialResultados****: List<ResultadoClustering>* con análisis ejecutados

Métodos principales:

- ***agregarPregunta(pregunta)****:* Añade pregunta al final de la lista.
- ***modificarPregunta(pregunta, index)****:* Reemplaza pregunta en posición específica. Valida índice antes de operar.
- ***eliminarPregunta(index)****:* Elimina pregunta de posición específica. Valida índice.
- ***eliminarTodasPreguntas()****:* Limpia completamente la lista de preguntas.
- ***getIndicePregunta(pregunta)****:* Busca pregunta por ID y retorna su índice. Retorna -1 si no existe.
- ***agregarResultado(resultado)****:* Añade resultado de clustering al historial.
- ***getUltimoResultado()****:* Retorna último clustering ejecutado o null.
- ***limpiarHistorial()****:* Elimina todos los resultados de clustering guardados.

Getters: id, titulo, descripcion, preguntas, numPreguntas, pregunta(index), historialResultados

Setters: titulo, descripcion, id (usado en deserialización)

2.1.12. Pregunta (abstracta)

Responsabilidad: Clase base para todos los tipos de preguntas. Define comportamiento común y contrato para subclases.

Atributos:

- **id**: Identificador único generado con UUID (String)
- **enunciado**: Texto de la pregunta (String)
- **obligatoria**: Indica si la respuesta es requerida (boolean)

Métodos abstractos:

- ***getTipoPregunta()****:* Retorna enum TipoPregunta (NUMERICA, CATEGORIA_SIMPLE, etc.)
- ***validarRespuesta(Object respuesta)****:* Verifica si el valor proporcionado es válido para este tipo de pregunta

Métodos concretos:

- ***setEnunciado(enunciado)****:* Actualiza enunciado validando que no esté vacío
- ***setObligatoria(obligatoria)****:* Marca pregunta como obligatoria/opcional
- ***setId(id)****:* Establece ID (usado en deserialización)
- Getters: id, enunciado, esObligatoria()

2.1.13. Numerica

Responsabilidad: Pregunta que acepta valores numéricos con rango opcional.

Hereda de: Pregunta

Atributos:

- **min**: Valor mínimo permitido (Double, puede ser null = sin límite inferior)
- **max**: Valor máximo permitido (Double, puede ser null = sin límite superior)

Constructores:

- ***Numerica(enunciado, min, max)****:* Con rango específico. Valida que min <= max
- ***Numerica(enunciado)****:* Sin restricción numérica (min y max null)

Validación: Verifica que respuesta sea null (válido si no obligatoria) o número. Si es Number lo convierte a Double. Si es String intenta parsearlo. Verifica rango: valor >= min y valor <= max. Rechaza NaN e Infinity.

Getters/Setters: min, max, setRango(min, max)

2.1.14. CategoriaSimple

Responsabilidad: Pregunta donde el usuario selecciona UNA opción de un conjunto predefinido.

Hereda de: Pregunta

Atributos:

- **opciones**: Set<String> con opciones válidas (no vacío)

Constructor: Valida que opciones no sea null ni esté vacío. Crea copia defensiva del Set.

Métodos:

- ***agregarOpciones(nuevaOpcion)****:* Añade nueva opción al conjunto
- ***eliminarOpcion(opcion)****:* Elimina opción del conjunto

Validación: Verifica que respuesta sea String y que esté contenida en el conjunto de opciones. null válido si no obligatoria.

Getters/Setters: *getOpciones()* (retorna copia), *setOpciones()*

2.1.15. CategoriaMultiple

Responsabilidad: Pregunta donde el usuario selecciona MÚLTIPLES opciones de un conjunto, con límite de selecciones.

Hereda de: Pregunta

Atributos:

- ***opciones****: Set<String>* con opciones válidas
- ***maxSelecciones***: Número máximo de opciones que se pueden seleccionar (int, mínimo 1)

Constructor: Valida maxSelecciones > 0. Permite opciones null (crea Set vacío).

Validación: Verifica que respuesta sea Set. Comprueba que no exceda maxSelecciones. Valida que cada elemento sea String y esté en opciones. Set vacío válido si no obligatoria.

Getters/Setters: opciones, maxSelecciones (valida >= 1)

2.1.16. Ordinal

Responsabilidad: Pregunta con opciones que tienen un orden implícito (ej: "Bajo", "Medio", "Alto"). Similar a CategoriaSimple pero el orden importa para clustering.

Hereda de: Pregunta

Atributos:

- ***opciones****: Set<String> ordenado* (LinkedHashSet mantiene orden de inserción)

Constructor: Crea copia del Set de opciones preservando orden.

Validación: Verifica que respuesta sea String contenida en opciones. null válido si no obligatoria.

2.1.17. Libre

Responsabilidad: Pregunta de texto abierto con límite de caracteres.

Hereda de: Pregunta

Atributos:

- **longitudMaxima**: Límite de caracteres permitidos (int, default 1000)

Constructores:

- **Libre(enunciado)**: Longitud máxima por defecto (1000)
- **Libre(enunciado, longitudMaxima)**: Con límite específico

Validación: Verifica que longitud del texto no exceda longitudMaxima. Usa toString() para obtener texto. null válido si no obligatoria.

2.1.18. Respuesta

Responsabilidad: Representa una respuesta individual de un usuario a una pregunta específica de una encuesta. Almacena el valor respondido junto con los identificadores necesarios para su trazabilidad completa.

Atributos:

- idUsuario: Identificador del usuario que proporciona la respuesta (String). Permite rastrear quién respondió.
- idPregunta: Identificador de la pregunta respondida (String). Vincula la respuesta con su pregunta origen.
- idEncuesta: Identificador de la encuesta a la que pertenece la pregunta (String). Proporciona contexto completo.

- valor: Contenido de la respuesta (Object). Tipo dinámico que puede ser: Double (preguntas numéricas), String (categoría simple, ordinal, libre), Set<String> (categoría múltiple), o null (pregunta no contestada).
- contestada: Indicador booleano de si la pregunta ha sido efectivamente respondida (boolean). Diferencia entre "no respondida" y "respondida con valor null".

Constructores:

- Respuesta(idUsuario, idPregunta, idEncuesta, valor): Constructor completo que inicializa todos los identificadores y el valor. Marca automáticamente contestada = true si el valor no es null.
- Respuesta(idUsuario, idPregunta, idEncuesta): Constructor para respuesta no contestada. Establece valor = null y contestada = false.

Métodos principales:

- limpiar(): Resetea la respuesta a su estado inicial no contestado. Establece valor = null y contestada = false. Útil para permitir que usuarios modifiquen sus respuestas.
- equals(Object obj): Compara respuestas por sus tres identificadores (idUsuario, idPregunta, idEncuesta). Dos respuestas son iguales si pertenecen al mismo usuario, pregunta y encuesta, independientemente de su valor. Permite usar Respuesta en colecciones HashSet y como clave en HashMap.
- hashCode(): Genera código hash basado en los tres identificadores. Consistente con equals() para correcto funcionamiento en colecciones.

Getters:

- getIdUsuario(): Retorna ID del usuario
- getIdPregunta(): Retorna ID de la pregunta
- getIdEncuesta(): Retorna ID de la encuesta
- getValor(): Retorna el valor de la respuesta (Object)
- isContestada(): Retorna boolean indicando si fue contestada

Setters:

- setValor(Object valor): Actualiza el valor de la respuesta. Automáticamente actualiza contestada según si el nuevo valor es null o no.
- setIdUsuario(String idUsuario): Permite modificar ID de usuario (usado principalmente en deserialización)
- setIdPregunta(String idPregunta): Permite modificar ID de pregunta
- setIdEncuesta(String idEncuesta): Permite modificar ID de encuesta

Uso típico: La clase Respuesta actúa como contenedor inmutable de información de respuesta. Se crea una instancia cuando un usuario responde una pregunta mediante ControladorRespuestas.responderPregunta(). Se almacena dentro de UsuarioRespondedor.respuestasPorEncuesta organizadas por encuesta. Durante clustering, los valores se extraen para formar la matriz de datos de análisis.

2.1.18 ResultadoClustering

Responsabilidad: Encapsula todos los datos resultantes de una ejecución de clustering, incluyendo asignaciones, centroides, métricas de calidad y metadata del proceso. Permite almacenar, comparar y visualizar resultados de diferentes ejecuciones.

Atributos principales:

Datos de agrupación:

- *groups:* Array de enteros (int[]) donde cada posición representa un usuario y el valor es el cluster asignado (0-based). Ejemplo: [0, 1, 0, 2] indica usuario0→cluster0, usuario1→cluster1, usuario2→cluster0, usuario3→cluster2. La longitud coincide con el número de usuarios analizados.
- centers: Matriz bidimensional (Object[][]) de dimensiones [k][numPreguntas] que almacena los centroides (KMeans) o medoides (KMedoids). Cada fila representa un cluster, cada columna una pregunta analizada. Los valores son del tipo apropiado: Double para numéricas, String para categóricas, Set<String> para múltiples.
- *idsUsuarios:* Lista ordenada (List<String>) con los IDs de usuarios en el mismo orden que el array groups. Permite mapear posiciones del array a usuarios reales. Esencial para presentación de resultados.

Métricas de calidad:

- *silhouette:* Coeficiente de silueta (double) en rango [-1, 1]. Mide qué tan bien están separados los clusters. Valores cercanos a 1 indican excelente separación, cercanos a -1 indican mala agrupación, alrededor de 0 indican clusters solapados. Se usa como criterio principal para seleccionar el mejor resultado.
- *inercia:* Suma de cuadrados de distancias intra-cluster (double). Mide compactación de los clusters. Valores menores indican clusters más compactos. Útil para método del codo.

Metadata del proceso:

- *algoritmo:* Nombre del algoritmo usado (String): "KMeans", "KMeans++", o "KMedoids". Permite identificar el método que generó el resultado.
- *k*: Número de clusters generados (int). Parámetro fundamental del clustering.
- *numIteraciones:* Número de iteraciones ejecutadas hasta convergencia (int). Indica cuántos ciclos asignación-recálculo fueron necesarios. Útil para diagnóstico de convergencia.
- *idEncuesta:* Identificador de la encuesta analizada (String). Vincula el resultado con su contexto.
- *indicePregunta:* Índice de la pregunta utilizada para el clustering (int). Permite saber qué pregunta se analizó.

Constructores:

- *ResultadoClustering(groups, centers, silhouette, idsUsuarios, algoritmo, k):* Constructor principal que requiere datos esenciales. Inicializa métricas opcionales en 0.
- *ResultadoClustering(groups, centers, silhouette, idsUsuarios, algoritmo, k, numIteraciones, inercia, idEncuesta, indicePregunta):* Constructor completo con toda la metadata. Usado al crear resultados desde ejecuciones de clustering.

Métodos principales:

- *getUsuariosPorGrupo():* Organiza los IDs de usuarios agrupados por cluster. Retorna Map<Integer, List<String>> donde la clave es el número de cluster (0-based) y el valor es la lista de IDs de usuarios asignados a ese cluster. Itera sobre el array groups acumulando usuarios en listas según su cluster. Útil para mostrar miembros de cada cluster en la UI.
- *getNumUsuariosEnGrupo(int grupo):* Cuenta cuántos usuarios fueron asignados al cluster especificado. Itera sobre groups contando coincidencias. Retorna int con el conteo. Útil para verificar distribución de tamaños de clusters.
- *toString():* Genera representación textual legible del resultado. Incluye: algoritmo, k, Silhouette, inercia, iteraciones. Lista usuarios por cluster llamando a getUsuariosPorGrupo(). Formatea centroides mostrando valores por pregunta. Retorna String multilínea formateado.

Getters completos:

- *getGroups():* Retorna copia del array de asignaciones
- *getCenters():* Retorna referencia a matriz de centroides (cuidado: mutable)
- *getSilhouette():* Retorna coeficiente de silueta
- *getIdsUsuarios()*: Retorna lista de IDs (copia defensiva recomendada en producción)
- *getAlgoritmo():* Retorna nombre del algoritmo
- *getK():* Retorna número de clusters
- *getNumIteraciones():* Retorna iteraciones ejecutadas
- *getInercia():* Retorna métrica de inercia
- *getIdEncuesta():* Retorna ID de encuesta analizada
- *getIndicePregunta():* Retorna índice de pregunta analizada
- **Setters:** Disponibles para todos los atributos para permitir modificación post-creación y facilitar deserialización desde JSON. En uso normal, los resultados son tratados como inmutables después de la creación.

Uso típico: Se crea una instancia al finalizar cada ejecución de clustering en Clustering.ejecutarClustering(). Se retorna a CtrlClustering que lo compara con otros resultados para seleccionar el mejor. El resultado óptimo se añade al historial de la encuesta y se serializa a JSON para persistencia. La UI puede recuperarlo para mostrar visualizaciones de clusters.

2.1.19. Clustering

Responsabilidad: Clase core del módulo de análisis que prepara los datos de respuestas para algoritmos de clustering y coordina su ejecución. Realiza transformación de objetos de dominio a matrices numéricas, maneja valores faltantes mediante imputación KNN, y configura estrategias de clustering con información de tipos de preguntas.

Atributos: Ninguno (clase sin estado). Todos los datos se pasan como parámetros para permitir ejecuciones paralelas y facilitar testing.

Métodos principales:

- ***ejecutarClustering(estrategia, usuarios, encuesta, indicePregunta)***

Método orquestador que coordina todo el proceso de clustering de principio a fin.

Parámetros:

- estrategia: Instancia de AlgoritmoClustering (KMeans, KMeansPlusPlus o KMedoids) configurada con k y maxIter
- usuarios: List<UsuarioRespondedor> con los usuarios que han respondido la encuesta
- encuesta: Objeto Encuesta con las preguntas y estructura
- indicePregunta: Índice (int) de la pregunta a analizar en el clustering

Proceso de ejecución:

1. **Preparación de datos:** Llama a *prepararDatos()* para extraer respuestas de los usuarios y convertirlas en matriz *Object[][]*. Cada fila representa un usuario, cada columna una pregunta. Los IDs de usuarios se almacenan en lista de salida para mantener correspondencia.
2. **Imputación de valores faltantes:** Si algún usuario no respondió alguna pregunta, llama a *imputarValoresNull()* que completa valores usando KNN con k=5 vecinos. Esto es crítico porque los algoritmos de clustering no pueden procesar valores null.
3. **Configuración de estrategia:** Llama a *configurarEstrategia()* para pasar a la estrategia información sobre tipos de preguntas, rangos numéricos y opciones ordinales. Esto permite que los algoritmos calculen distancias apropiadas según el tipo.
4. **Ejecución del algoritmo:** Delega a *estrategia.execute(datos)* que retorna el objeto *ResultadoClustering*.
5. **Enriquecimiento del resultado:** Añade metadata al resultado: idEncuesta, indicePregunta, numIteraciones. Establece los IDs de usuarios para permitir mapeo.

Retorna: *ResultadoClustering* completo con todos los datos y métricas.

Excepciones: Puede lanzar IllegalArgumentException si la estrategia es null, la encuesta no tiene preguntas, o el índice de pregunta es inválido.

- ***prepararDatos(usuarios, encuesta, indicePregunta, outIds)***

Método privado que transforma las respuestas almacenadas en objetos Usuario en una matriz de datos lista para clustering.

Parámetros:

- usuarios: Lista de UsuarioRespondedor con respuestas
- encuesta: Encuesta con estructura de preguntas
- indicePregunta: Índice de la pregunta a analizar
- outIds: Lista de salida (List<String>) donde se almacenarán los IDs en orden

Proceso:

1. Valida que haya usuarios y preguntas disponibles
2. Obtiene número de preguntas de la encuesta
3. Crea matriz Object[numUsuarios][numPreguntas]
4. Para cada usuario:
    - Añade su ID a outIds (mantiene orden usuario↔fila)
    - Obtiene sus respuestas a la encuesta mediante *getRespuestasEncuesta()*
    - Crea mapa de respuestas por ID de pregunta para acceso O(1)
    - Para cada pregunta de la encuesta:
        - Busca respuesta del usuario en el mapa
        - Si existe y está contestada, extrae el valor
        - Si no existe o no está contestada, asigna null
        - Almacena en matriz[usuarioIdx][preguntaIdx]

Retorna: Matriz *Object[][]* lista para imputación y clustering. Puede contener valores null que serán manejados después.

Nota de implementación: El uso de un mapa intermedio por usuario optimiza la búsqueda de respuestas de O(n×m) a O(n+m), crucial con muchas preguntas.

- ***imputarValoresNull(datos, encuesta)***

Método privado que completa valores faltantes en la matriz usando imputación KNN (K-Nearest Neighbors).

Parámetros:

- datos: Matriz Object[][] con posibles valores null
- encuesta: Encuesta con tipos de preguntas para cálculo de distancias

Proceso:

1. Itera sobre cada celda de la matriz (usuario × pregunta)
2. Detecta valores null que requieren imputación
3. Para cada valor null encontrado:
    - Llama a imputarConKNN() con las coordenadas
    - Asigna el valor imputado de vuelta a la matriz
4. Continúa hasta procesar toda la matriz

Retorna: void (modifica matriz in-place)

Estrategia de imputación: KNN con k=5 es un compromiso entre robustez (k alto reduce efecto de outliers) y localidad (k bajo mantiene patrones locales). El valor 5 es estándar en literatura de machine learning.

- ***imputarConKNN(usuarioIdx, preguntaIdx, datos, encuesta)***

Método privado que calcula el valor a imputar para una celda específica usando promedio ponderado de los 5 usuarios más similares.

Parámetros:

- usuarioIdx: Fila del valor a imputar (int)
- preguntaIdx: Columna del valor a imputar (int)
- datos: Matriz completa de datos
- encuesta: Encuesta para cálculo de distancias

Proceso:

1. Crea lista de pares (índiceUsuario, distancia) para todos los usuarios excepto el actual
2. Para cada otro usuario:
    - Calcula distancia usando calcularDistanciaParcial() que ignora la pregunta con valor null
    - Almacena par (índice, distancia)
3. Ordena lista por distancia ascendente (más similares primero)
4. Selecciona los k=5 vecinos más cercanos
5. Obtiene el tipo de la pregunta a imputar
6. **Para preguntas NUMERICAS:**
    - Calcula promedio ponderado: suma(valor_vecino / distancia_vecino) / suma(1/distancia_vecino)
    - Si distancia es 0 (usuario idéntico), usa su valor directamente
    - Retorna Double
7. **Para preguntas categóricas (CATEGORIA_SIMPLE, ORDINAL):**
    - Cuenta frecuencia de cada valor entre los vecinos
    - Pondera frecuencias por 1/distancia (vecinos más cercanos tienen más peso)
    - Retorna la categoría con mayor peso acumulado (moda ponderada)
8. **Para CATEGORIA_MULTIPLE:**
    - Acumula todos los Sets de vecinos
    - Cuenta frecuencia de cada opción

    - Retorna Set con opciones más frecuentes (umbral: aparece en >50% de vecinos)
9. **Para LIBRE:**
    - Usa moda simple (valor más frecuente)
    - Ignora distancias (difícil ponderar texto)

Retorna: Object con valor imputado del tipo apropiado (Double, String, Set<String>)

Manejo de casos especiales:

- Si todos los vecinos tienen null en esa pregunta, usa valor por defecto del tipo (0.0 para numéricas, "Sin respuesta" para categóricas)
- Si hay empate en moda categórica, usa el primero encontrado (orden alfabético por estabilidad)

- ***calcularDistanciaParcial(usuario1, usuario2, preguntaIgnorada, datos, encuesta)***

Método privado auxiliar que calcula la distancia entre dos usuarios excluyendo una pregunta específica. Esencial para imputación KNN.

Parámetros:

- usuario1: Índice del primer usuario (int)
- usuario2: Índice del segundo usuario (int)
- preguntaIgnorada: Índice de la pregunta a excluir del cálculo (int)
- datos: Matriz de datos completa
- encuesta: Encuesta para obtener tipos de preguntas

Proceso:

1. Inicializa acumulador de distancia total y contador de preguntas válidas
2. Para cada pregunta en la encuesta:
    - Si es la pregunta ignorada, salta a siguiente iteración
    - Si alguno de los dos usuarios tiene null en esa pregunta, salta (no contribuye a distancia)
    - Llama a *calcularDistanciaUnaPregunta()* para esa pregunta
    - Acumula distancia y incrementa contador
3. Calcula promedio: distanciaTotal / numPreguntasComparadas
4. Si no hay preguntas en común (contador=0), retorna Double.MAX_VALUE (máxima distancia posible)

Retorna: double con distancia promedio normalizada en [0,1]

Justificación del promedio: Normalizar por número de preguntas permite comparar usuarios con diferente cantidad de respuestas válidas. Un usuario con 5 preguntas en común es comparable a uno con 8.

- ***calcularDistanciaUnaPregunta(respuesta1, respuesta2, pregunta)***

Método privado que calcula la distancia entre dos respuestas a una pregunta específica, usando métrica apropiada según el tipo.

Parámetros:

- respuesta1: Valor de la primera respuesta (Object)
- respuesta2: Valor de la segunda respuesta (Object)
- pregunta: Objeto Pregunta con tipo y metadata

Proceso según tipo:

NUMERICA:

- Extrae valores como Double
- Obtiene rango: max - min de la pregunta
- Si rango es 0 (valor constante), retorna 0.0
- Calcula: |valor1 - valor2| / rango
- Normaliza a [0,1] dividiendo por el rango total
- Ejemplo: Pregunta con min=0, max=100. Respuestas 30 y 80 → |30-80|/100 = 0.5

ORDINAL:

- Convierte respuestas a String
- Obtiene lista ordenada de opciones de la pregunta
- Encuentra posición de cada respuesta en la lista
- Calcula: |pos1 - pos2| / (numOpciones - 1)
- Normaliza a [0,1] por número de opciones
- Ejemplo: Opciones ["Bajo","Medio","Alto"]. "Bajo" vs "Alto" → |0-2|/(3-1) = 1.0

CATEGORIA_SIMPLE:

- Distancia binaria simple
- Retorna 0.0 si respuestas son iguales (usando .equals())
- Retorna 1.0 si respuestas son diferentes
- Ejemplo: "Rojo" vs "Rojo" → 0.0, "Rojo" vs "Azul" → 1.0

CATEGORIA_MULTIPLE:

- Convierte respuestas a Set<String>
- Calcula índice de Jaccard: |A ∩ B| / |A ∪ B|
- Intersección: opciones presentes en ambos sets
- Unión: todas las opciones presentes
- Retorna: 1.0 - Jaccard (convertir similitud en distancia)
- Ejemplo: A={a,b,c}, B={b,c,d} → Jaccard=2/4=0.5 → distancia=0.5

LIBRE:

- Convierte respuestas a String
- Calcula distancia de Levenshtein (mínimo número de ediciones para transformar un string en otro)
- Normaliza dividiendo por la longitud del string más largo
- Retorna valor en [0,1]
- Ejemplo: "hola" vs "hala" → Levenshtein=1, max=4 → 1/4 = 0.25

Retorna: double en rango [0,1] donde 0=idénticas, 1=máximamente diferentes

Nota importante: Todas las distancias están normalizadas a [0,1] para permitir promediarlas equitativamente sin que un tipo de pregunta domine el cálculo de distancia global.

- ***configurarEstrategia(estrategia, encuesta)***

Método privado que transfiere información de tipos de preguntas desde la encuesta a la estrategia de clustering.

Parámetros:

- estrategia: Instancia de *AlgoritmoClustering* a configurar
- encuesta: *Encuesta* con las preguntas

Proceso:

1. Crea array de TipoPregunta con tamaño = número de preguntas
2. Para cada pregunta en la encuesta:
    - Obtiene tipo mediante getTipoPregunta()
    - Almacena en array en posición correspondiente
3. Pasa array a estrategia mediante setTipoPreguntas()
4. Para cada pregunta:
    - Si es NUMERICA: extrae min/max y llama setNumericRange(index, min, max)
    - Si es ORDINAL: extrae lista de opciones ordenadas y llama setOrdinalOptions(index, options)

Retorna: void (configura estrategia in-place)

Propósito: Permite que los algoritmos calculen distancias correctamente. Sin esta información, no sabrían si normalizar valores numéricos o aplicar Jaccard a categóricas.

- ***configurarSemilla(estrategia, semilla)***

Método privado que establece la semilla del generador aleatorio de la estrategia para reproducibilidad.

Parámetros:

- estrategia: Instancia de AlgoritmoClustering
- semilla: Valor long para inicializar Random

Proceso: Llama a estrategia.setSeed(semilla) que inicializa el generador aleatorio interno.

Importancia: Permite ejecutar el mismo clustering múltiples veces con resultados idénticos, esencial para debugging y para el proceso de selección de mejor resultado donde se hacen 10 ejecuciones con diferentes semillas.

2.1.20. AlgoritmoClustering (interfaz)

Responsabilidad: Define el contrato que deben cumplir todos los algoritmos de clustering del sistema. Abstrae las diferencias de implementación permitiendo tratar KMeans, KMeans++ y KMedoids polimórficamente.

Métodos obligatorios:

- execute(data)

- **Parámetro:** data - Matriz Object[][] con datos preparados e imputados
- **Retorna:** ResultadoClustering con asignaciones, centros y métricas
- **Propósito:** Ejecuta el algoritmo completo hasta convergencia o máximo de iteraciones
- **Comportamiento esperado:**
    - Inicializar centros/medoides según estrategia del algoritmo
    - Iterar: asignar usuarios a clusters → recalcular centros
    - Detectar convergencia (asignaciones estables)
    - Calcular métricas finales (Silhouette, inercia)
    - Retornar resultado completo

- setTipoPreguntas(tipos)

- **Parámetro:** tipos - Array de TipoPregunta[]
- **Retorna:** void
- **Propósito:** Configura qué tipo de dato contiene cada columna de la matriz
- **Uso:** El algoritmo usa esta información en calculateDistance() para aplicar métricas apropiadas

- setNumericRange(index, min, max)

- **Parámetros:** index (int), min (double), max (double)
- **Retorna:** void
- **Propósito:** Establece el rango válido de una columna numérica para normalización
- **Uso:** Permite calcular (valor - min) / (max - min) para normalizar distancias numéricas

- setOrdinalOptions(index, options)

- **Parámetros:** index (int), options (List<String>)
- **Retorna:** void
- **Propósito:** Proporciona el orden de las opciones de una pregunta ordinal
- **Uso:** Permite calcular distancia entre opciones según posición: |indexOf(A) - indexOf(B)|

- setSeed(seed)

- **Parámetro:** seed - long para inicializar Random
- **Retorna:** void
- **Propósito:** Establece semilla para reproducibilidad de inicializaciones aleatorias
- **Uso:** Garantiza que ejecuciones con misma semilla produzcan resultados idénticos

2.1.21. KMeans, KMeansPlusPlus, KMedoids

Implementan: *AlgoritmoClustering*

Atributos: *k, maxIter, questionTypes, minValues, maxValues, ordinalOptions, random*

Métodos comunes:

- *execute(data):* Bucle asignación→recálculo centros hasta convergencia
- *calculateDistance(personA, personB):* Distancia promedio entre respuestas normalizada [0,1]
- *calculateSilhouette(data, groups):* Coeficiente de separación clusters
- *calcularInercia(data, groups, centers):* Suma cuadrados distancias intra-cluster
- *setSeed(seed):* Reproducibilidad

Diferencias:

- **KMeans:** Inicialización aleatoria de centros
- **KMeansPlusPlus:** Inicialización inteligente (K-Means++)
- **KMedoids:** Usa puntos reales como centros, encontrarMejoresMedoides minimiza distancia total, kmedoidspp para inicialización

Distancias según tipo:

- **NUMERICA:** |A-B|/(max-min)
- **ORDINAL:** |posA-posB|/(n-1)
- **CATEGORIA_SIMPLE:** 0 si igual, 1 si diferente
- **CATEGORIA_MULTIPLE:** 1 - Jaccard(A,B)
- **LIBRE:** Levenshtein normalizada

2.1.22. AdaptadorPregunta

Implementa: IndicePregunta

Responsabilidad: Conectar Clustering con CtrlDominio para obtener índices de preguntas

2.2. Capa de Persistencia

2.2.1. CtrlPersistencia

Responsabilidad: Fachada única de persistencia que coordina todas las operaciones de almacenamiento.

Atributos:

- *persistenciaEncuestas, persistenciaUsuarios, persistenciaRespuestas, persistenciaClustering, persistenciaCSV:* Delegados especializados
- *encuestasCargadas, usuariosCargados, respuestasCargadas:* Flags para carga bajo demanda

Funcionalidad:

- **Inicialización:** Crea estructura de directorios, carga datos bajo demanda (lazy loading)
- **Delegación:** Todos los métodos delegan a persistencias especializadas
- **Encuestas:** CRUD de JSON de encuestas (guardarEncuesta, obtenerEncuesta, listarEncuestas, eliminarEncuesta)
- **Usuarios:** Gestión separada de admins/respondedores en JSON (guardarRespondedor, guardarAdmin, obtenerUsuario, esAdmin)
- **Respuestas:** CRUD de respuestas en CSV único (guardarRespuestas, obtenerRespuestas, obtenerUsuariosQueRespondieron)
- **Clustering:** Persistencia de resultados JSON por encuesta (guardarResultadosClustering, obtenerResultadosClustering)
- **CSV:** Lectura/escritura de archivos CSV genéricos (leerCSV, escribirCSV)
- **Caché inteligente:** Solo carga datos cuando se acceden por primera vez (asegurarEncuestasCargadas)

2.2.2. PersistenciaEncuestas

Responsabilidad: Gestión de archivos JSON individuales de encuestas.

Atributos:

- encuestasJSON: Map<String, String> (id → JSON)
- directorioEncuestas: Ruta ./data/encuestas/

Funcionalidad:

- Guarda cada encuesta en archivo individual: {id}.json
- *guardar(encuestaJSON):* Extrae ID del JSON, almacena en mapa y persiste archivo con pretty printing
- *obtener(id):* Retorna JSON desde caché
- *cargar():* Lee todos los archivos .json del directorio al iniciar
- *eliminar(id):* Borra del caché y elimina archivo físico
- **Ventaja:** Cada encuesta es independiente, facilita backups y versionado

2.2.3. PersistenciaUsuarios

Responsabilidad: Persistencia de usuarios en dos archivos JSON separados.

Atributos:

- respondedoresJSON, adminsJSON: Map<String, String> (id → JSON)
- archivoRespondedores, archivoAdmins: ./data/usuarios/{respondedores.json, administradores.json}

Funcionalidad:

- **Separación por tipo:** Admins y respondedores en archivos distintos
- *guardarRespondedor/Admin(usuarioJSON):* Extrae ID, actualiza caché, persiste archivo
- *obtener(id):* Busca primero en admins, luego en respondedores
- *cargarRespondedores/Admins():* Lee JsonArray y puebla caché
- *esAdmin(id):* Verifica existencia en caché de admins
- **Formato:** Array JSON con todos los usuarios del mismo tipo

2.2.4. PersistenciaRespuestas

Responsabilidad: Almacenamiento de respuestas en CSV único.

Atributos:

- cache: Map<idEncuesta, Map<idUsuario, Map<idPregunta, valor>>> (triple nivel)
- archivoRespuestas: ./data/respuestas/todas_respuestas.csv
- lector, escritor: Componentes CSV

Funcionalidad:

- **Formato CSV:** Columnas: ID_Encuesta | ID_Usuario | ID_Pregunta | Valor
- *guardarRespuestas(idUsuario, idEncuesta, respuestas):* Actualiza caché y persiste todo el archivo
- *obtenerRespuestas(idUsuario, idEncuesta):* Extrae desde caché triple nivel
- *obtenerUsuariosQueRespondieron(idEncuesta):* Lista usuarios con respuestas en encuesta
- *persistirTodo():* Convierte todo el caché a filas CSV y sobrescribe archivo
- *cargar():* Lee CSV completo

2.2.5. PersistenciaClustering

Responsabilidad: Almacenamiento de resultados de clustering por encuesta.

Atributos:

- resultadosJSON: Map<String, String> (idEncuesta → JSON de resultados)
- directorioClustering: ./data/clustering/

Funcionalidad:

- Archivo por encuesta: clustering_{idEncuesta}.json
- *guardarResultados(idEncuesta, resultadosJSON):* Persiste lista de ResultadoClustering como JSON
- *obtenerResultados(idEncuesta):* Retorna JSON String desde caché
- *existenResultados(idEncuesta):* Verifica si hay resultados guardados
- *eliminarResultados(idEncuesta):* Borra del caché y elimina archivo
- *cargarTodos():* Lee todos los archivos clustering_*.json al iniciar
- **Uso:** Historial de ejecuciones de clustering con diferentes K y algoritmos

2.2.6. PersistenciaCSV

Responsabilidad: Coordinador de lectura/escritura CSV (fachada).

Atributos:

- lector: LectorCSV
- escritor: EscritorCSV

Funcionalidad:

- *leerArchivo(ruta):* Retorna List<String[]> (matriz de datos crudos)
- *leerConEncabezados(ruta):* Retorna DatosCSV con encabezados separados y FilaRespuesta
- *escribirArchivo(ruta, filas):* Escribe matriz de strings a CSV
- *escribirConEncabezados(ruta, encabezados, filas):* Combina encabezados con datos
- **Clases internas:**
    - DatosCSV: Estructura con encabezados (String[]) y filas (List<FilaRespuesta>)
    - FilaRespuesta: Wrapper con idUsuario y respuestas[] (separa primera columna)
- **Conversión:** Transforma entre formatos de LectorCSV y PersistenciaCSV

2.2.7. LectorCSV

Responsabilidad: Lectura y parsing de archivos CSV.

Atributos:

- separador: String (default ",")
- saltarLineasVacias: boolean (default true)

Funcionalidad:

- *leerArchivo(rutaArchivo):* Lee CSV completo, retorna List<String[]>
- *leerConEncabezados(rutaArchivo):* Primera fila → encabezados, resto → FilaRespuesta
- *parsearLinea(linea):* Split por separador, limpia espacios
- *validarRuta(rutaArchivo):* Verifica existencia, tipo archivo, permisos lectura
- **Clases internas:**
    - *FilaRespuesta*: idUsuario + respuestas[] (columna 0 separada del resto)
    - *DatosCSV*: Contenedor de encabezados y lista de FilaRespuesta con métodos de acceso
- **Robustez:** Manejo de FileNotFoundException, IOException, líneas vacías

2.2.8. EscritorCSV

Responsabilidad: Escritura y formateo de archivos CSV.

Atributos:

- separador: String (default ",")
- incluirBOM: boolean (default false, para UTF-8)

Funcionalidad:

- *escribirArchivo(rutaArchivo, filas):* Escribe List<String[]> a CSV
- *formatearLinea(valores):* Convierte String[] a línea CSV con separadores
- *escaparValor(valor):* Maneja comillas, saltos de línea, valores con comas (duplica comillas internas, envuelve en comillas)
- *validarRuta(rutaArchivo):* Crea directorios padre si no existen
- **Formato:**
    - Valores con coma/comilla/newline → envueltos en comillas
    - Comillas internas → duplicadas (" → "")
    - Null → string vacío
- **Opcional:** BOM para compatibilidad Excel UTF-8

2.2.9. PreguntaTypeAdapter

Responsabilidad: Adaptador Gson para serialización/deserialización polimórfica de Pregunta.

Implementa: JsonSerializer<Pregunta>, JsonDeserializer<Pregunta>

Funcionalidad:

- **serialize():**
    - Añade campo tipoPregunta para identificar tipo
    - Guarda campos comunes: id, enunciado, obligatoria
    - Serializa campos específicos según tipo:
        - NUMERICA: min, max
        - CATEGORIA_SIMPLE/ORDINAL: opciones
        - CATEGORIA_MULTIPLE: opciones, maxSelecciones
        - LIBRE: longitudMaxima
- **deserialize():**
    - Lee tipoPregunta para determinar tipo
    - Crea instancia concreta según TipoPregunta (Numerica, CategoriaSimple, etc.)
    - Restaura campos comunes (setId, setObligatoria)
    - Deserializa campos específicos con TypeToken para Set<String>

Uso: Registrado en Gson para manejar polimorfismo de Pregunta sin perder información de tipo

2.3. Capa de Presentación

2.3.1. CtrlPresentacion

Responsabilidad: Controlador central de la capa de presentación que gestiona la navegación enrte vistas y mantiene el estado de sesión.

Atributos:

- primaryStage: Ventana principal de JavaFX
- ctrlDominio: Referencia al controlador de dominio
- usuarioActualId: ID del usuario con sesión activa
- esAdmin: Flag que indica si el usuario es administrador

Funcionalidad:

- **Inicialización:** Configura ventana principal (tamaño, título, límites)
- **Navegación:** Métodos para mostrar cada vista (mostrarLogin, mostrarMenuPrincipal, mostrarCrearEncuesta, etc.)
- **Gestión de sesión:** Almacena usuario actual, cierra sesión, determina tipo de usuario
- **Cambio de escena:** Método privado cambiarEscena() que actualiza Stage con nueva Scene
- **Acceso:** Getters para que las vistas accedan al controlador de dominio, usuario actual y Stage

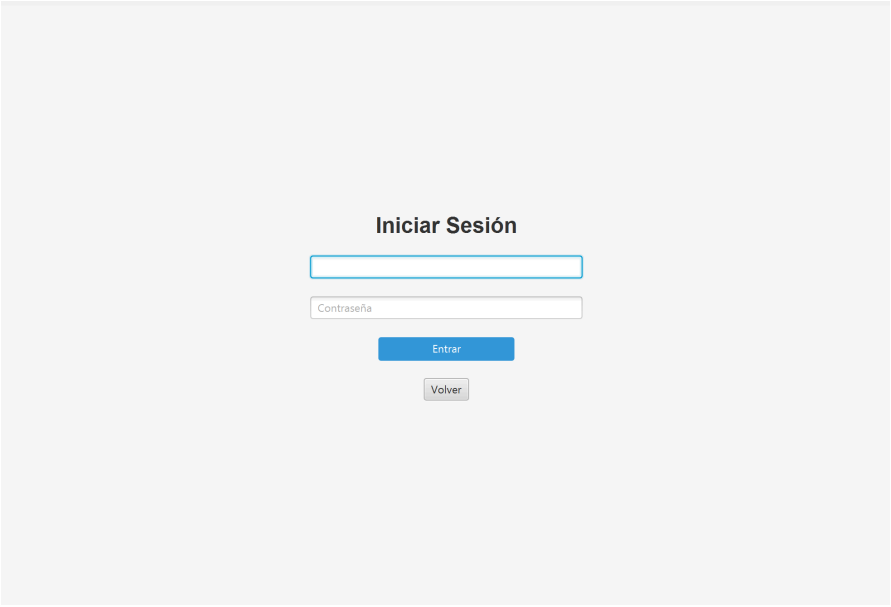
2.3.2. MenuInicialView

Responsabilidad: Pantalla de bienvenida del sistema.

Funcionalidad:

- Muestra título y subtítulo del sistema
- Botones para registro de administrador/respondedor
- Botón para login
- Botón para salir de la aplicación
- Estilos visuales con colores diferenciados por acción
- Efectos hover en botones

2.3.2. LoginView

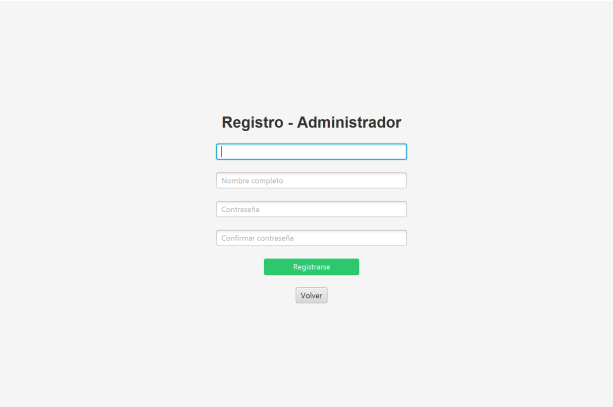


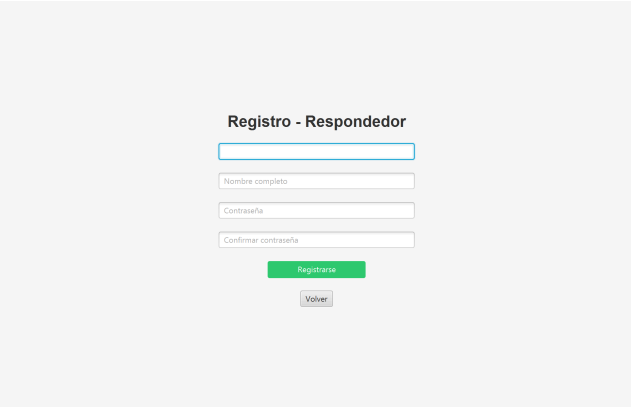
Responsabilidad: Pantalla de inicio de sesión.

Funcionalidad:

- Campos: ID de usuario y contraseña
- Validación de campos no vacíos
- Obtiene usuario del dominio mediante ID
- Determina si es admin o respondedor (instanceof UsuarioAdmin)
- Redirige al menú correspondiente según tipo de usuario
- Botón volver al menú inicial
- Muestra alertas de error si usuario no existe o campos vacíos

2.3.3. RegistroView





Responsabilidad: Formulario de registro de nuevos usuarios.

Atributos:

- esAdmin: Flag para determinar tipo de registro

Funcionalidad:

- Campos: ID, nombre, contraseña, confirmar contraseña
- Validación: campos completos y contraseñas coincidentes
- Llama a crearUsuarioAdmin() o crearUsuarioRespondedor() según tipo
- Muestra confirmación y redirige a login tras éxito
- Manejo de excepciones con alertas de error

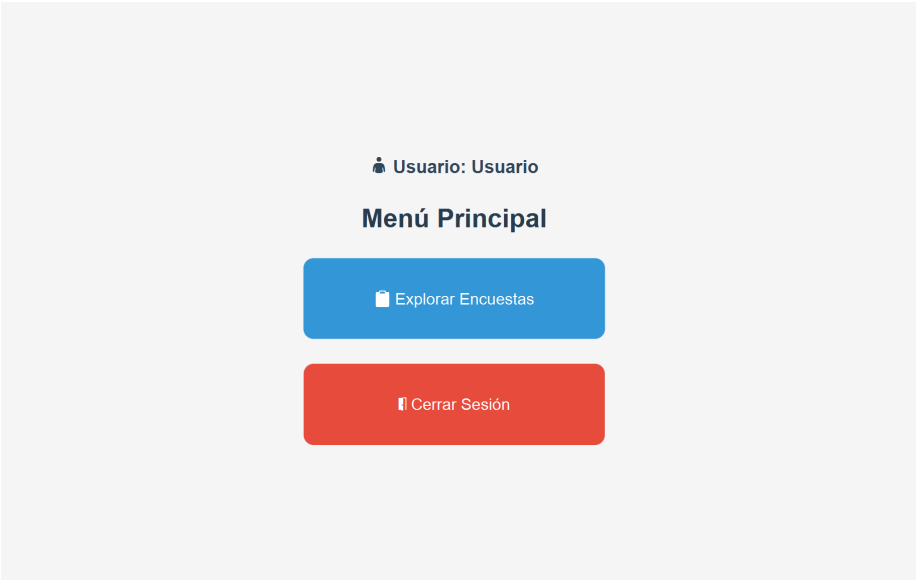
2.3.4. MenuAdminView

Responsabilidad: Menú principal para usuarios administradores.

Funcionalidad:

- Muestra nombre del usuario actual
- Secciones organizadas:
    - **Explorar:** Listar encuestas
    - **Gestión:** Crear nueva encuesta
    - **Importar/Exportar:** Importar CSV, Exportar CSV
- Botón cerrar sesión
- Botones con estilos diferenciados por sección y efectos hover

2.3.5. MenuRespondedorView

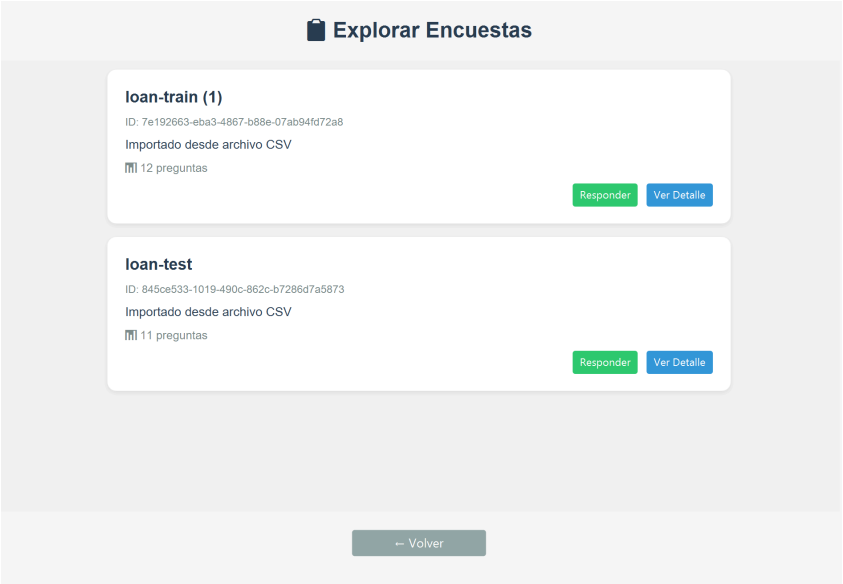


Responsabilidad: Menú principal para usuarios respondedores.

Funcionalidad:

- Muestra nombre del usuario actual
- Botón para explorar encuestas disponibles
- Botón cerrar sesión
- Interfaz simplificada (sin opciones de administración)
- Estilos visuales consistentes con el sistema

2.3.6. ListarEncuestasView



Responsabilidad: Vista de exploración de todas las encuestas.

Funcionalidad:

- Obtiene lista de encuestas del dominio
- Muestra mensaje si no hay encuestas
- Crea card visual por cada encuesta con:
    - Título, ID, descripción
    - Número de preguntas
    - Botones según tipo de usuario:
        - **Admin:** Ver detalle, Gestionar preguntas, Estadísticas
        - **Respondedor:** Ver detalle, Responder (si hay preguntas)
- Efectos hover en cards
- ScrollPane para lista larga
- Botón volver al menú correspondiente

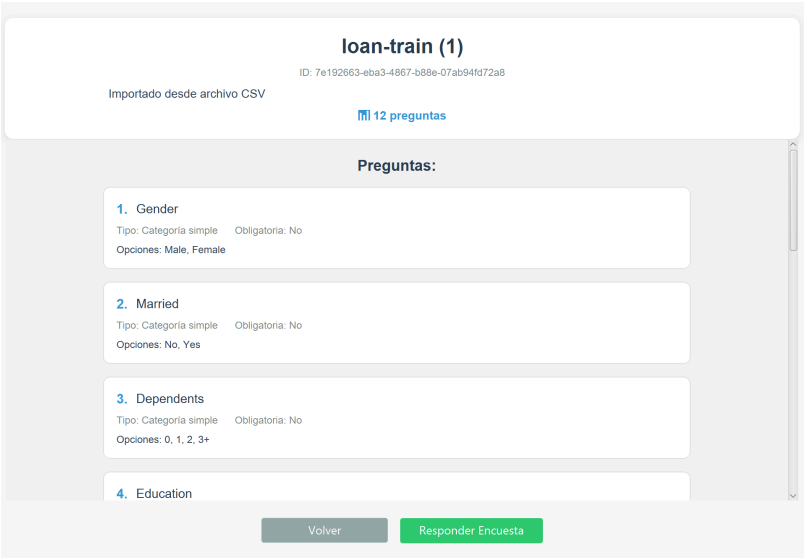
2.3.7. CrearEncuestaView

Responsabilidad: Formulario para crear nuevas encuestas.

Funcionalidad:

- Campos: Título (obligatorio), Descripción
- Validación de título no vacío
- Llama a crearEncuesta() del dominio
- Dialog de confirmación tras crear con opciones:
    - Añadir preguntas inmediatamente
    - Volver al menú
- Botón cancelar que vuelve al menú admin

2.3.8. DetalleEncuestaView



Responsabilidad: Vista detallada de una encuesta específica.

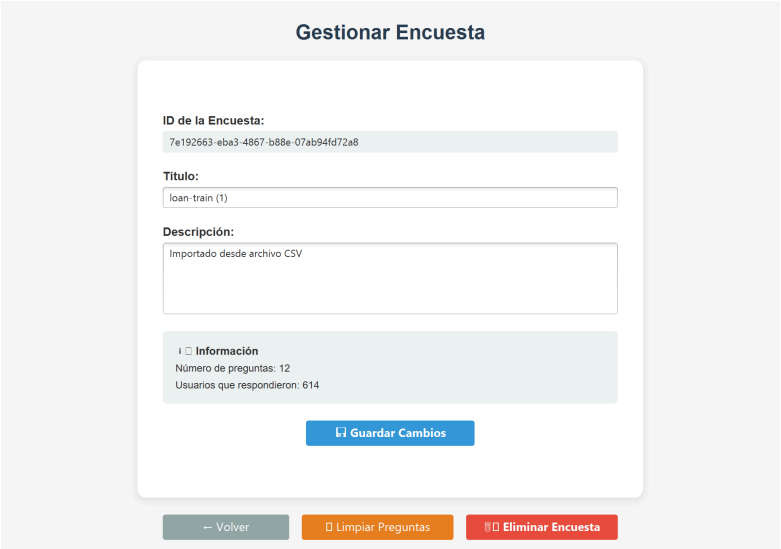
Atributos:

- idEncuesta: ID de la encuesta a mostrar

Funcionalidad:

- **Header:** Título, ID, descripción, número de preguntas
- **Centro:** Lista de cards con cada pregunta mostrando:
  - Número y enunciado
  - Tipo de pregunta
  - Obligatoriedad
  - Detalles específicos (rango, opciones, límites)
- **Footer - Botones según usuario:**
  - **Todos:** Volver
  - **Admin:** Gestionar encuesta, Gestionar preguntas, Ver estadísticas
  - **Respondedor:** Responder (si hay preguntas)
- Método obtenerNombreTipo() para mostrar tipo legible
- Método obtenerDetallesPregunta() para info específica por tipo

2.3.9. GestionarEncuestaView



Responsabilidad: Edición y eliminación de encuestas.

Atributos:

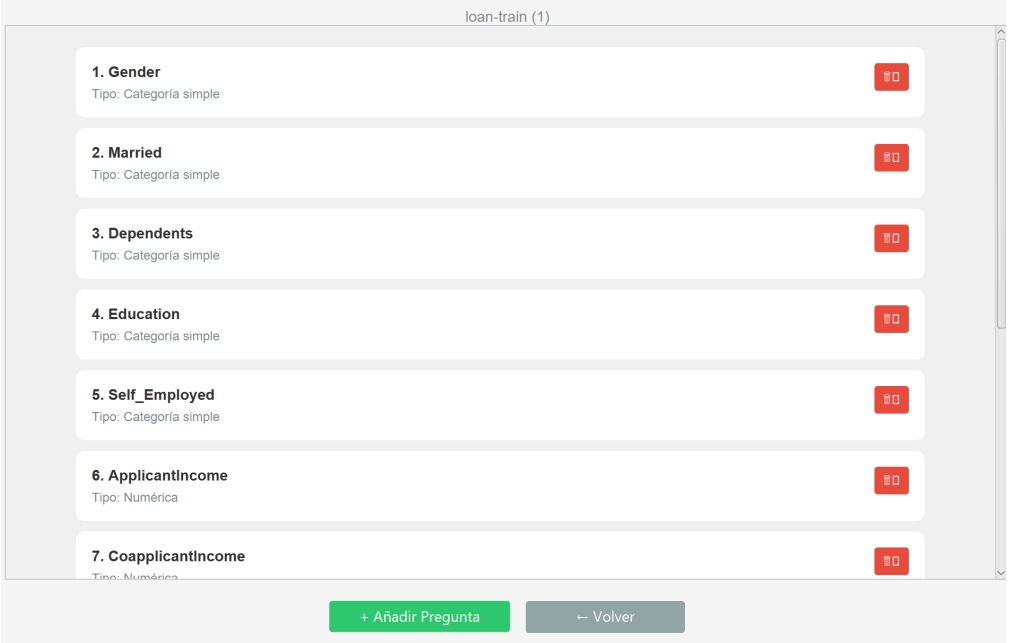
- idEncuesta: ID de la encuesta a gestionar

Funcionalidad:

- **Formulario:**
  - ID (solo lectura)
  - Título (editable)
  - Descripción (editable)
  - Info: número de preguntas y respuestas

- **Acciones:**
    - Guardar cambios: llama a modificarEncuesta()
    - Limpiar preguntas: elimina todas las preguntas de atrás hacia adelante
    - Eliminar encuesta: con confirmación, elimina encuesta completa
- Confirmaciones para acciones destructivas
- Validación de título no vacío
- Recarga vista tras modificaciones

2.3.10. GestionarPreguntasView



Responsabilidad: Gestión de preguntas de una encuesta.

Atributos:

- idEncuesta: ID de la encuesta

Funcionalidad:

- Lista actual de preguntas con número, enunciado, tipo
- Botón eliminar por pregunta (con confirmación)
- **Botón añadir pregunta** abre diálogo con:
    - ComboBox tipo de pregunta
    - TextField enunciado
    - CheckBox obligatoria
- **Diálogos específicos según tipo:**
    - **Numérica:** min/max opcionales
    - **Libre:** spinner longitud máxima
    - **Categoría Simple/Ordinal:** TextArea para opciones (una por línea)
    - **Categoría Múltiple:** TextArea opciones + spinner max selecciones
- Llama a addPregunta() del dominio
- Recarga vista tras cada cambio

2.3.11. ResponderEncuestaView

Responsabilidad: Formulario para responder encuestas.

Atributos:

- idEncuesta: ID de la encuesta a responder
- controlesRespuesta: Map<idPregunta, Node> para almacenar controles

Funcionalidad:

- **Crea controles según tipo de pregunta:**
  - **Numérica:** TextField
  - **Libre:** TextArea
  - **Categoría Simple/Ordinal:** ComboBox
  - **Categoría Múltiple:** VBox con CheckBoxes
- **Validación:** Verifica preguntas obligatorias
- **Extracción de valores:**
  - TextField: String o Double según tipo
  - TextArea: String
  - ComboBox: valor seleccionado
  - VBox CheckBoxes: Set<String> de seleccionados
- Llama a responderEncuesta() del dominio con Map<idPregunta, valor>
- Muestra confirmación y vuelve a lista tras enviar

2.3.12. EstadisticasView

Responsabilidad: Visualización de estadísticas de una encuesta.

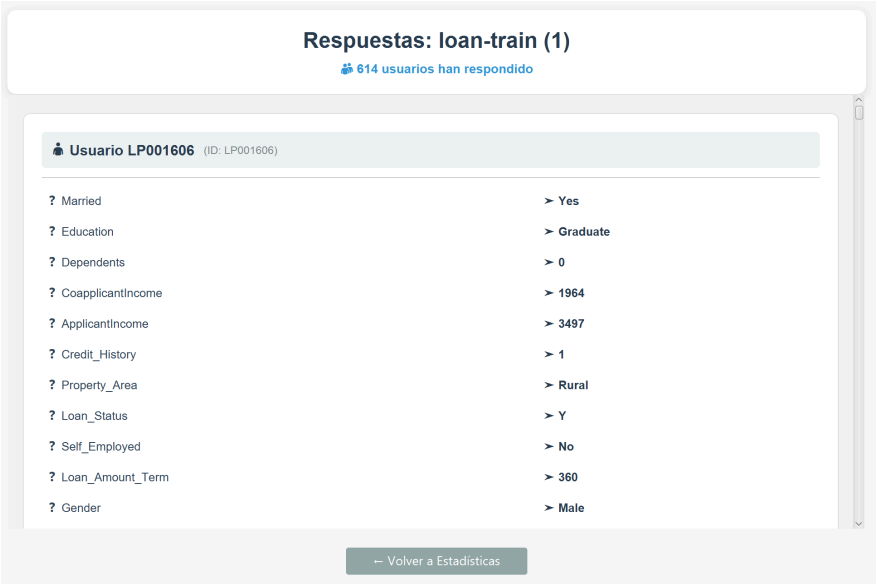
Atributos:

- idEncuesta: ID de la encuesta

Funcionalidad:

- **Resumen general:** Cards con número de preguntas y respuestas
- **Estadísticas por pregunta:**
    - **Numérica:** Media, mínimo, máximo
    - **Categoría Simple/Ordinal:** Frecuencias con barras de progreso
    - **Categoría Múltiple:** Frecuencias de cada opción seleccionada
    - **Libre:** Longitud promedio
- Método calcularEstadisticasPregunta(): itera respuestas y calcula stats según tipo
- Método crearBarraFrecuencia(): ProgressBar + porcentaje
- Método formatearNumero(): sin decimales si es entero
- Botones: Volver, Ver respuestas detalladas, Análisis de clustering

2.3.13. VerRespuestasView



Responsabilidad: Vista detallada de respuestas individuales por usuario.

Atributos:

- idEncuesta: ID de la encuesta

Funcionalidad:

- Obtiene usuarios que respondieron
- Crea card por usuario con:
    - Header: nombre e ID
    - Lista de respuestas: pregunta → respuesta
- Método formatearValor():
    - Set → valores separados por comas
    - Double → sin decimales si es entero
    - null → "(Sin respuesta)"
- Botón volver a estadísticas

2.3.14. ClusteringView

Responsabilidad: Configuración, ejecución y visualización de clustering.

Atributos:

- idEncuesta: ID de la encuesta
- cmbAlgoritmo, spinnerK, spinnerIteraciones: Controles de configuración
- contenedorResultados: VBox para mostrar resultados

Funcionalidad:

- **Panel configuración:**
  - ComboBox algoritmo (K-Means, K-Means++, K-Medoids)
  - Spinner K máximo (2-10)
  - Spinner iteraciones (10-1000)
- **Ejecutar clustering:**
  - Verifica respuestas disponibles
  - Valida K ≤ número usuarios
  - Ejecuta en hilo separado para no bloquear UI
  - Llama a ejecutarClustering() del dominio
  - Muestra progreso con Alert
- **Visualización resultados:**
  - Carga automática de historial guardado
  - Identifica y destaca mejor resultado (mayor Silhouette)
  - Cards por resultado con: K, algoritmo, Silhouette
  - Distribución: usuarios por cluster con ProgressBar
  - Usa getUsuariosPorGrupo() de ResultadoClustering
  - Diálogo detalles: lista completa de usuarios por cluster
- **Limpiar historial:** Elimina todos los resultados guardados con confirmación
- Botones: Volver a estadísticas, Menú principal

2.3.15. ImportarCSVView

Responsabilidad: Importación de encuestas desde archivos CSV.

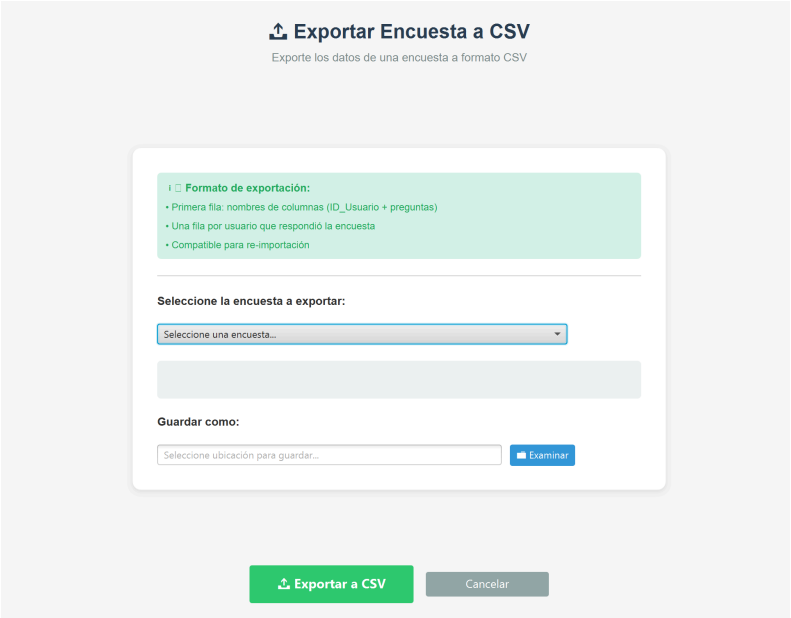
Atributos:

- txtRuta, txtTitulo, txtDescripcion: Campos del formulario
- txtVistaPrevia: TextArea para preview
- btnImportar: Botón de acción

Funcionalidad:

- **Instrucciones formato CSV:**
  - Primera fila: nombres columnas
  - Primera columna: ID usuario
  - Resto: respuestas (tipos inferidos)
- **Selección archivo:** FileChooser con filtro .csv
- **Vista previa:**
  - Lee CSV con LectorCSV del dominio
  - Muestra encabezados y primeras 10 filas
  - Formato tabular con separador |
- **Importación:**
  - Valida archivo y título
  - Ejecuta en hilo separado
  - Llama a importarCSV() del dominio
  - Dialog de progreso durante importación
  - Confirmación con opciones: Ver encuesta o Volver
- Auto-completa título con nombre de archivo

2.3.16. ExportarCSVView



Responsabilidad: Exportación de encuestas a archivos CSV.

Atributos:

- cmbEncuestas: ComboBox con EncuestaItem
- txtRuta: Campo destino
- lblInfo: Label información exportación
- btnExportar: Botón de acción

Funcionalidad:

- **Instrucciones formato exportación:**
  - Primera fila: ID_Usuario + nombres preguntas
  - Una fila por usuario respondedor
  - Compatible para re-importación
- **Selección encuesta:**
  - Filtra solo encuestas con preguntas y respuestas
  - Obtiene InfoExportacion del dominio
  - ComboBox muestra: título (preguntas, respuestas)
- **Actualización info:**
  - Muestra preguntas, usuarios, dimensiones archivo resultante
  - Auto-sugiere nombre archivo basado en título
- **Selección destino:** FileChooser con extensión .csv
- **Exportación:**
  - Ejecuta en hilo separado
  - Llama a exportarEncuesta() del dominio
  - Dialog progreso durante exportación
  - Confirmación con opción abrir ubicación
- **Clase interna EncuestaItem:** Wrapper para mostrar encuesta con info en ComboBox

3. Estructuras de datos y algoritmos utilizados

3.1 Estructuras de datos

3.1.1 List y ArrayList

En nuestro proyecto se utiliza la interfaz **List** y su implementación más común, **ArrayList**. Este tipo de estructura aparece, por ejemplo, en:

- La lista de preguntas de una encuesta → *Encuesta.getPreguntas()*
- Las listas de respuestas de un usuario a una encuesta → *Map<String, List<Respuesta>>* en *UsuarioAdmin* y *UsuarioRespondedor*
- Las filas leídas de un CSV en *LectorCSV* → *List<String[]>*
- La colección de usuarios o encuestas devuelta por los gestores: *listarUsuarios*, *listarEncuestas*.
- El almacenamiento temporal de filas válidas en *GestorClustering.prepararDatos*: lista de *Object[]* posteriormente convertida a *Object[][]*

Se ha optado por **ArrayList** porque:

- Permite acceso por índice en **O(1)**.
- Ofrece buen rendimiento en recorridos secuenciales y accesos por posición.
- La inserción al final también es **O(1)**.
- No se necesita insertar en el medio ni mantener el orden explícitamente.

En este sistema, las listas se usan como:

- Contenedores de objetos de dominio (**preguntas**, **encuestas**, **usuarios**).
- Estructuras temporales para construir matrices de datos utilizadas en *clustering*.
- Representaciones directas de filas provenientes de un CSV.

3.1.2 Map y HashMap

La estructura clave del dominio es **Map**, implementado mediante **HashMap**. Se utiliza para:

- Guardar encuestas por identificador en ***GestorEncuestas***
  Map<String, Encuesta> encuestas
- Guardar usuarios por identificador en ***GestorUsuarios***
  Map<String, UsuarioRespondedor> y *Map<String, UsuarioAdmin>*
- Asociar a cada encuesta la lista de respuestas de un usuario
  Map<String, List<Respuesta>> respuestasPorEncuesta
- Mapear preguntas con respuestas al exportar a **CSV**
  Map<String, Respuesta> interno en *CtrlCSV.exportarEncuesta*
- Representar conjuntos gestionados por **ID** con acceso rápido

Ventajas de usar **HashMap**:

- Búsqueda y actualización en **O(1)** promedio
- Simplifica la lógica del código evitando recorridos secuenciales
- Reduce acoplamiento y aumenta legibilidad, usando simplemente *map.get(id)*

3.1.3 Set, HashSet y LinkedHashSet

Se utiliza la interfaz **Set** mediante dos implementaciones:

- **HashSet**, por ejemplo:
  - En preguntas de tipo *Ordinal* para definir opciones válidas → *Set<String> opciones*
  - En respuestas de categoría múltiple → *Set<String> opcionesSeleccionadas*

- **LinkedHashSet**, usado en:
  - CtrlCSV.extraerOpcionesPorColumna para recolectar opciones del CSV manteniendo orden

Finalidades principales de los *sets*:

1. **Definir dominios de valores válidos**
   - Utilizado en preguntas *ORDINAL*, *CATEGORIA_SIMPLE* y *CATEGORIA_MULTIPLE*

2. **Representar respuestas con múltiples opciones**
   - Evita duplicados de forma automática

La elección de **LinkedHashSet** asegura que el orden original del CSV se mantenga en la interfaz final.

3.1.4 Arrays y matrices (String[], Object[], Object[][], int[])

Aunque el dominio se basa en colecciones dinámicas, se utilizan **arrays** en partes numéricas:

- En *LectorCSV*
  - Cada fila → *String[]*
  - Cabeceras → *String[]*

- En *ResultadoClustering*
  - Grupos por usuario → *int[] groups*
  - Centros de clúster → *Object[][] centers*

- En *GestorClustering*
  - *prepararDatos()* genera una matriz *Object[][]* donde cada fila es un usuario y cada columna una respuesta

Motivos de uso:

- Acceso por índice **rápido** y **directo**
- Tamaño fijo → apropiado para vectores de características
- Facilita cálculos algorítmicos y numéricos

3.1.5 Otras estructuras

- ***List<List<String>>*** en *ResultadoClustering.getUsuariosPorGrupo*
  Para agrupar IDs por clúster.
- ***enum TipoPregunta***
  Para modelar tipos válidos de pregunta evitando errores por *strings*.
- ***UUID*** en Pregunta
  Para generar IDs únicos y controlados incluso tras importación de datos.

3.2 Algoritmos

3.2.1 Gestión de usuarios, encuestas y preguntas

En los gestores (*GestorUsuarios, GestorEncuestas*) se implementan algoritmos básicos de alta, baja y modificación:

- **Creación de encuestas**
  GestorEncuestas.crearEncuesta valida que el título no esté vacío, instancia la encuesta y la guarda en el *Map<String, Encuesta>*.
  Complejidad: **O(1)** para la inserción en el mapa.
- **Modificación de encuestas**
  modificarEncuesta comprueba si la encuesta existe y, si es así, actualiza título y/o descripción solo si no son *null*.
- **Gestión de preguntas**
  - *addPregunta* añade la pregunta a la lista interna de la encuesta.
  - *modificarPregunta* comprueba que el índice sea válido antes de sustituir una pregunta por otra.
  - *eliminarPregunta* encapsula la lógica de borrado de una pregunta concreta, capturando posibles *IndexOutOfBoundsException*.

En GestorUsuarios se sigue un patrón similar:

- Antes de crear un usuario se comprueba con *existeUsuario(id)* que no haya otro con el mismo identificador.
- La obtención de usuarios que han respondido a una encuesta (*obtenerUsuariosQueRespondieron*) recorre todos los usuarios y consulta en cada uno si tiene respuestas para ese id de encuesta. Es un algoritmo lineal respecto al número de usuarios.

3.2.2 Validación de respuestas por tipo de pregunta

La clase abstracta *Pregunta* define el contrato *boolean validarRespuesta(Object respuesta)*, que cada subtipo implementa según su semántica.

- Preguntas numéricas (*Numerica*)

La validación sigue los pasos:

1. Si valor es *null*, solo se acepta si la pregunta no es obligatoria (*!esObligatoria()*).
2. Si el valor es un *Number*, se convierte a *double*.
3. Si es un *String*, se hace un *trim()* y se intenta *parsear* con *Double.valueOf*. Si lanza *NumberFormatException*, la respuesta no es válida.
4. Se descartan explícitamente valores *NaN* o *infinitos*.
5. Si hay **mínimos** y **máximos** definidos (min y max no nulos), se comprueba que el valor esté dentro del rango. En caso contrario se lanza una excepción (*IllegalArgumentException*) que permite detectar datos inconsistentes.

Este algoritmo garantiza que todas las respuestas numéricas que llegan al sistema sean realmente números y estén dentro de los límites definidos al crear la pregunta.

- Preguntas de texto libre (*Libre*)

En *Libre.validarRespuesta*:

1. Si el valor es *null*, se acepta solo si la pregunta no es obligatoria.
2. En caso contrario, se convierte a *String* y se comprueba que su longitud no supere *longitudMaxima*.

Es un algoritmo sencillo de coste **O(n)** en la longitud del texto, suficiente para controlar que el usuario no escriba respuestas excesivamente largas.

- Preguntas ordinales (*Ordinal*)

La validación de una respuesta ordinal consiste en:

1. Tratar *null* y cadenas vacías como “no responde” y aceptar solo si la pregunta no es obligatoria.
2. Comprobar que el valor sea un *String*.
3. Verificar que *opciones.contains(valor)*.

El uso de un *Set<String>* hace que esta comprobación sea **O(1)** promedio. La lógica de orden (la semántica de “más alto” o “más bajo”) se utiliza posteriormente en el *clustering*, pero la validación se limita a garantizar que la opción existe.

Otras preguntas categóricas (simple y múltiple), aunque no se muestran aquí, siguen una idea similar: comprobar pertenencia a un conjunto de opciones y, en el caso de selección múltiple, comprobar que el número de opciones marcadas no supere el máximo permitido.

3.2.3 Algoritmos de respuesta a encuestas

La clase *GestorRespuestas* centraliza la lógica de registro de respuestas:

- *responderPregunta(Usuario usuario, Encuesta encuesta, String idPregunta, Object valor)*:
    1. Localiza la pregunta dentro de la encuesta usando *encontrarPregunta*.
    2. Llama a *pregunta.validarRespuesta(valor)*. Si devuelve *false*, se lanza una **excepción**.
    3. Crea un objeto *Respuesta* con ids de usuario, pregunta y encuesta.
    4. Registra esta respuesta en el propio usuario mediante *usuario.addRespuesta*.

El coste de este algoritmo es **lineal** en el número de preguntas de la encuesta, ya que la búsqueda de la pregunta se hace recorriendo la lista (*for (Pregunta p : encuesta.getPreguntas())*).

- *responderEncuesta(Usuario usuario, Encuesta encuesta, Map<String, Object> respuestasPorPregunta*):
    1. Primero comprueba que todas las preguntas obligatorias tengan una entrada en el mapa de respuestas.
    2. Después recorre las entradas del mapa y va llamando a *responderPregunta* para cada pregunta, capturando posibles errores para no abortar todo el proceso.

Este algoritmo es **lineal** en el número de respuestas que llegan en el mapa, y se apoya en la validación específica de cada tipo de pregunta.

3.2.4 Importación y exportación de encuestas en CSV

El controlador *CtrlCSV* encapsula dos algoritmos importantes: **exportación** e **importación**.

Exportación *(exportarEncuesta)*

A grandes rasgos, el proceso es:

1. Recuperar la encuesta y sus preguntas.
2. Buscar todos los usuarios que han respondido a esa encuesta.
3. Construir la primera fila del CSV con los tipos de pregunta (NUMERICA, ORDINAL, etc.).
4. Para cada usuario:
    - Se construye un mapa *idPregunta → Respuesta* para acceder rápido.
    - Se recorre la lista de preguntas en orden, rellenando un *String[]* con el valor formateado de cada respuesta mediante *formatearRespuestaParaCSV*.
5. Finalmente, se llama a EscritorCSV para volcar los encabezados y todas las filas en el fichero.

El algoritmo recorre usuarios × preguntas, es decir, su coste es **O(U·P)** donde U es el número de usuarios que han respondido y P el número de preguntas.

Importación *(importarCSV)*

La importación es algo más elaborada:

1. Se lee el archivo CSV con *LectorCSV.leerConEncabezados*, obteniendo:
    - Una primera fila con los tipos de pregunta.
    - Una lista de filas con las respuestas de cada usuario.
2. Se crea una nueva Encuesta con el título y descripción proporcionados.
3. Se llama a *extraerOpcionesPorColumna*:
    - Para cada columna se recorre todas las filas y se guardan los valores distintos en un *LinkedHashSet*.
    - En las columnas de tipo CATEGORIA_MULTIPLE se separan los valores por comas para extraer todas las opciones posibles.
    - El resultado es una lista de conjuntos de opciones reales por columna.
4. Se construyen las preguntas reales a partir de los tipos y de las opciones calculadas en el paso anterior:
    - Para NUMERICA se calcula el mínimo y máximo de los valores presentes y se da un pequeño margen.
    - Para tipos ordinales y categóricos se crean los conjuntos de opciones directamente a partir de los valores diferentes que aparecen en el CSV.
    - Para LIBRE se toma como referencia la longitud máxima observada y se le añade un margen.
5. Se procesan las filas de respuestas (*procesarRespuestasCSV*):
    - Para cada fila se crea un usuario respondedor “ficticio” (*user_csv_i*).
    - Se convierten los valores de la fila al tipo adecuado según la pregunta (*convertirValor*).
    - Se registran las respuestas usando *gestorRespuestas.responderPregunta*.

Este algoritmo también tiene coste **O(U·P),** pero con cierta sobrecarga inicial para calcular opciones y rangos a partir de los datos.

Además, el método *validarCSV* permite comprobar antes de importar que:

- Hay al menos una columna y una fila de datos.
- Todos los tipos de pregunta en la primera línea pertenecen al conjunto permitido.

3.2.5 Preparación de datos para el clustering

El componente *GestorClustering* se encarga de transformar las respuestas en una representación numérica adecuada para los algoritmos de agrupamiento:

1. Construcción de la matriz de datos (*prepararDatos*)
    - Recorre la lista de usuarios que han respondido.
    - Para cada usuario:
        - Llama a *convertirRespuestasAArray*, que:
            - Obtiene las respuestas del usuario en esa encuesta.
            - Para cada respuesta, calcula el índice de la pregunta con *obtenerIndicePregunta*.
            - Inserta el valor en la posición adecuada del array.
        - Si alguna posición queda a *null* (es decir, el usuario no respondió todas las preguntas), se utiliza un algoritmo de *Machine Learning* no supervisado llamado *KNN* que es capaz de rellenar los *nulls* con valores que se asemejan a los k puntos (o en este caso respuestas) más cercanos.
    - El resultado final es un *Object[][]* donde:
        - Cada fila representa un usuario.
        - Cada columna representa una pregunta.
    - Paralelamente se guarda la lista de ids de usuario en el mismo orden para poder reconstruir los grupos después.

2. Configuración del algoritmo (*configurarAlgoritmo*)
    - Se recorre la lista de preguntas de la encuesta:
        - Si la pregunta es Numerica, se llama a *clustering.configurarRangoNumerico* con los valores **min** y **max**.
        - Si la pregunta es Ordinal, se llama a *clustering.configurarOpcionesOrdinales* con el conjunto de opciones.
    - De esta forma, la lógica de distancia/similitud interna del algoritmo de clustering conoce el dominio de cada dimensión.
3. Extracción de tipos de pregunta (*extraerTiposPreguntas*)
    - Se construye un array *TipoPregunta[]* que se pasa al contexto de clustering para que el algoritmo sepa en cada dimensión si está tratando un número, una categoría, un ordinal o una respuesta libre.

3.2.6 Algoritmos de clustering y selección del mejor k

La clase *CtrlDominio* expone el método:

public *ResultadoClustering ejecutarClustering(String idEncuesta, String algoritmo, int k, int maxIter)*

Este método no solo lanza el *clustering*, sino que busca **el mejor número de grupos** entre 2 y k utilizando la medida de calidad *silhouette*:

1. Se obtiene la encuesta y la lista de usuarios que han respondido.
2. Para cada valor de i entre 2 y k:
    - Se crea una estrategia concreta (*KMeans, KMedoids o KMeans++*) a partir del string *algoritmo*.
    - Se instancia el contexto *Clustering* con esa estrategia.
    - Se llama a *gestorClustering.ejecutarClustering*, que devuelve un *ResultadoClustering*.
3. Se va guardando el resultado con mejor silhouette y al final se devuelve solo ese.

De este modo, el sistema no fija a priori un único valor de k, sino que explora varios y se queda con el que mejor agrupa a los usuarios según sus respuestas.

3.2.7 Generación de grupos de usuarios

Finalmente, la clase *ResultadoClustering* incluye un método auxiliar:

public List<List<String>> getUsuariosPorGrupo()

Este método reconstruye la estructura grupo → lista de usuarios de forma eficiente:

1. Crea una lista de listas, una por cada grupo (de 0 a k−1).
2. Recorre el array *groups*, que indica el número de grupo asignado a cada usuario.
3. Usa idsUsuarios.get(i) para recuperar el id del usuario correspondiente y lo añade a la lista de su grupo.

Esto facilita la presentación de los clústeres en la interfaz o su exportación a formatos como CSV.

4. Algoritmos de Clustering

4.1 Idea general

En este apartadose enseñarán los pseudocódigos simplificados de las distintas clases directamente relacionadas con clustering. En concreto, se describen las clases *ResultadoClustering*, *AlgoritmoClustering* y sus implementaciones (*KMeans*, *KMeansPlusPlus*, *KMedoids*), así como el gestor *Clustering*, que se encarga de preparar los datos y orquestar la ejecución de los algoritmos.

Las funcionalidades adicionales ligadas al preprocesado (KNN), búsqueda de la mejor k (método del codo) y métricas adicionales (inercia) se detallan en el subapartado 4.7.

4.2 ResultadoClustering

En esta clase encapsulamos y agrupamos todo el resultado del proceso ***clustering*** y sirve como transferencia de datos hacia la **capa de dominio** o de **presentación**.

Contenido:

- *int[] groups*
  Vector que indica, para cada usuario (fila del dataset), el clúster asignado.
- *Object[][] centers*
  - En *KMeans/KMeans++* → centroides.
  - En *KMedoids* → los usuarios que son medoides.
- *double silhouette*
  Silhouette medio del clustering final.
- double inercia
  Se utiliza como métrica adicional de calidad y en el método del codo
- *List<String> idsUsuarios*
  Mantiene el "mapeo" fila → id original del usuario, necesario para que la matriz *Object[][]* no pierda la información de **identidad**.
- Metadades:
  - Nombre del algorismo utilizado.
  - *k*
  - Número de iteraciones totales.

La clase *ResultadoClustering* actúa como transferencia de datos, no hace cálculos adicionales, solo encapsula el resultado que devuelve el algoritmo. El constructor garantiza la coherencia entre los arrays (por ejemplo, *groups.length()* coincida con *idsUsuarios.size()*), de manera que representa siempre al mismo usuario.

Métodos auxiliares

- *obtenerUsuariosPorGrupo()*

Devuelve una estructura *List<List<String>>* donde cada lista contiene los **IDs** de los usuarios que pertenecen a cada cluster, esto es útil para mostrar los resultados en una interfaz o incluso si queremos exportarlos a un **CSV**.

4.3 AlgoritmoClustering

La capa del clustering sigue el patrón **estrategia** que nos permite encapsular diferentes algorismos en una misma interfaz (*AlgoritmoClustering*). Esto facilita cambiar de algorismo sin modificar código externo y, además, añade extensibilidad para futuros métodos de clustering que queramos añadir.

Hay **tres** implementaciones: *KMeans*, *KMeansPlusPlus* y *KMedoids* junto a la clase auxiliar *ResultadoClustering*

La interfaz *AlgoritmoClustering* expone los siguientes métodos:

- *ResultadoClustering execute(Object[][] data):* ejecuta el algoritmo sobre el dataset ya preprocesado y devuelve el resultado completo.

- *void setTipoPreguntas(TipoPregunta[] tipos):* informa al algoritmo del tipo de cada columna para poder aplicar la distancia adecuada.

- *void setNumericRange(int questionIndex, double min, double max):* configura el rango de las preguntas numéricas para normalizar las distancias.

- *void setOrdinalOptions(int questionIndex, Set<String> options):* indica el orden de las categorías en preguntas ordinales.

De esta manera, la lógica de la distancia (numérica, ordinal, categórica, etc.) queda encapsulada dentro de los algoritmos de clustering y puede reutilizarse sin que la capa de dominio tenga que preocuparse por estos detalles.

4.4 KMeans

Estructuras de datos principales

1. *int k* → número de **clusters** deseados.
2. *int maxIter* → número máximo de **iteraciones** del proceso.
3. *Random random* → utilizado para la inicialización **aleatoria**.
   Este generador puede configurarse desde fuera mediante el método *setSeed(long seed)*, lo que permite reproducir siempre el mismo resultado de clustering si se usa la misma semilla.
4. *Object[][] centers* → matriz con los centroides (un vector para cada cluster).
5. *int[] groups* → para cada usuario, el índice del clúster al cual ha sido asignado.
6. Información adicional heredada de *AlgoritmoClustering*:
   - *TipoPregunta[] questionTypes*
   - Diccionarios con rangos números, opciones ordenadas, etc.

Algorismo detallado:

1. **Inicialización de los centroides**
   - Se escogen *k* usuarios aleatoriamente del dataset y sus respuestas se copian como centroides iniciales.
   - Esto como ventaja tiene que el **coste** es bajo
   - Como desventaja tiene que puede dar soluciones de **baja calidad** si la cantidad de puntos iniciales están muy juntos
2. **Bucle iterativo**
   Se repite hasta que
   - No hay cambios en las asignaciones o se ha llegado a *maxIter*

   La iteración tiene dos pasos:
   a. **Asignación a los clusters**
      - Para cada usuario se calcula la distancia a cada centro usando *calculateDistance*.
      - La distancia es mixta, depende el tipo de la pregunta:
        - Numérica: diferencia normalizada
        - Ordinal: diferencia de posiciones
        - Categoria simple: 0/1
        - Categoria múltiple: distancia tipo Jaccard
        - Texto libre: similitud basada en Levenshtein
      - El usuario asigna el centro más cercano → **O(k·m)** por usuario

   b. **Recalcular los centroides:**
      - Para cada cluster y por cada pregunta:
        - Pregunta numérica: se calcula la media
        - Pregunta categórica simple: se coge la moda (la opción más frecuente)
        - Ordinal: se coge el valor con posición en la mitad
        - Categoría múltiple: se coge la unión/intersección ponderada
        - Texto libre: se usa un "centroide simbólico", escogiendo el texto más representativo

- Los nuevos centros se guardan a centers
- Si un clúster queda vacío (no tiene ningún usuario asignado), el algoritmo reutiliza el centro anterior si existe o reescoge aleatoriamente una fila del dataset como nuevo centro. Esto evita que haya centroides "huérfanos" y garantiza que siempre haya exactamente k clusters activos.

3. **Criterio de parada**
   - Si *groups* no cambia en toda una iteración, se asume convergencia

4. **Evaluación con Silhouette**
   - Para cada usuario se calcula:
     - distancia media a su cluster (a)
     - distancia mínima a sus otros clusters (b)
     - silhouette = (b - a) / max(a, b)
   - La media de todos los silhouettes da la calidad global
   - Además, en esta fase también se calcula la inercia total del clustering. Ambos valores (silhouette media e inercia) se incluyen en el *ResultadoClustering* para poder comparar distintas ejecuciones y distintos valores de k.

Observaciones
- Complejidad general:
  O(maxIter · n · k · m)
- Muy eficiente en datasets grandes pero sensible a los puntos iniciales.
- Permite reproducir experimentos configurando una semilla aleatoria fija, lo que facilita la comparación entre distintos algoritmos o diferentes parámetros.

4.5 KMeansPlusPlus

Es una versión mejorada de *KMeans* que mantiene exactamente el mismo núcleo de asignación y recomputación, pero substituye la inicialización

Diferencias principales
- En lugar de escoger *k* muestras aleatorias, se usa una inicialización por probabilidad
- Para cada centro nuevo
  1. El primero se escoge **aleatoriamente**.
  2. Se calcula, para cada usuario, la **distancia al centro más cercano** ya seleccionado.
  3. Se escoge el siguiente centro con una probabilidad **proporcional** a la **distància²**.

Esto reduce drásticamente la probabilidad de obtener centroides muy juntos entre sí ya que tiende a colocar los primeros centros en zonas muy separadas del dataset. Por ende acostumbra a producir unas mejores soluciones y llegamos a necesitar menos iteraciones para dejar de tener cambios. El resto del bucle (asignación, recomputación, criterio de parada y silhouette) son idénticos al *KMeans* tradicional que hemos mencionado anteriormente.

Igual que en KMeans, al final de la ejecución se calculan tanto la silhouette media como la inercia y se empaquetan en un *ResultadoClustering*. La única diferencia real está en la inicialización de los centros, que añade un coste extra de **O(k·n)** pero suele reducir el número de iteraciones necesarias para converger y mejora la calidad de la solución final.

4.6 KMedoids

A diferencia de *KMeans*, que calcula centroides "virtuales", *KMedoids* selecciona medoides reales del dataset (son usuarios reales que representan el cluster). Con esto conseguimos dos cosas:

- Más robusto a los valores extremos.
- Más interpretables los centros (son usuarios existentes)

Estructuras de datos

- *int[] medoidIndices* → índices de los usuarios que son los medoides.
- *int[] groups* → asignación usuario → clúster.
- Comparteix amb *KMeans*:
    - *TipoPregunta[] questionTypes*
    - Diccionario de rangos y opciones
    - Función *calculateDistance*

Algorismo detallado

1. Inicialización de **medoides** con una variante de ***k-medoids***++.
    - Se escoge un medoide inicial aleatorio.
    - El resto se escogen parecido a *KMeans*++ pero asegurando que no existan duplicados, esto se hace gracias a un *Set<Integer>*.
    - Se reduce el riesgo de escoger dos medoides muy cercanos.

2. **Bucle principal**:

Como hemos comentado ya, hasta convergencia o *maxIter*.

a. **Asignación**
    - Cada usuario se asigna al medoide más cercano.
    - Equivalente a *KMeans* pero con medoides en vez de centroides.

b. **Actualización de los medoides**
    - Per a cada clúster:
        1. Se consideran todos sus miembros como posibles candidatos a nuevos medoides.
        2. Para cada candidato se calcula **suma**(distancias del candidato a todos los miembros del cluster).
        3. Se escoge el candidato que **minimiza** esta suma (**medoide óptimo local**).
    - Esto es más costoso que un *KMeans* pero es mucho más **robusto**.

c. **Gestión de clusters vacíos**
    - Si un cluster queda sin usuarios: se selecciona el nuevo medoide entre los puntos que no han sido utilizados hasta ahora, para evitar duplicados utilizamos un *Set<Integer>* que guarda todos los índices que ya están actuando como medoides.

3. **Cálculo de silhouette**
    - Lo hacemos igual que en *KMeans*:
        - Para cada usuario se calcula:
            1. distancia media a su cluster (a)
            2. distancia mínima a sus otros clusters (b)
            3. silhouette = (b - a) / max(a, b)
        - La media de todos los silhouettes da la calidad global
    - Se utiliza para dar una medida de la **calidad** del clustering final.

Observaciones

- Complejidad aproximada:
  O(maxIter · k · (n/k)² · m)
- Muy útil cuando
    - El dataset tiene valores extremos.
    - Las variables son categóricas o mixtas.
    - Se quieren centros interpretables
- Al igual que en las variantes de KMeans, la implementación de KMedoids calcula la silhouette media y la inercia total del clustering y las devuelve en el *ResultadoClustering*, lo que permite comparar directamente las diferentes estrategias sobre el mismo dataset.

4.7 Funcionalidades extras

KNN

Puesto que los algoritmos de clustering no pueden trabajar con valores null ya que tanto K-Means, K-Means++ como K-Medoids asumen que cada usuario tiene una respuesta válida para cada pregunta. Para resolver este problema, dentro del gestor de clustering (*Clustering*) usamos un algoritmo K-Nearest Neighbors (KNN) que imputa información en los null usando las instancias más parecidas a la que tenemos que rellenar.

En nuestra implementación, usamos un valor fijo *K_NEIGHBORS* = 5. Este proceso de imputación se ejecuta dentro del gestor Clustering antes de llamar a cualquier algoritmo de clustering, de modo que la matriz de datos que reciben *KMeans*, *KMeansPlusPlus* o *KMedoids* ya no contiene valores null.

El funcionamiento, a alto nivel, es el siguiente:

1. **Detección de nulos**
   Se recorre la matriz de respuestas *Object[][]* usuario × pregunta. Cada posición *datos[i][p]* que sea null se marca como candidata a imputación.

2. **Búsqueda de vecinos (K vecinos más cercanos)**
   Para cada respuesta nula de un usuario i en la pregunta p, el sistema:

   - Calcula la distancia entre el usuario i y el resto de usuarios, pero ignorando la pregunta que queremos imputar (*calcularDistanciaParcial*).

   - Esta distancia se calcula combinando las distancias pregunta a pregunta usando la información de la encuesta:

     - **NUMÉRICA**: distancia normalizada respecto al rango [min, max].
     - **ORDINAL**: distancia en base a la posición de cada categoría en la lista de opciones.
     - **CATEGORÍA_SIMPLE**: 0 si la respuesta coincide, 1 si no.
     - **CATEGORÍA_MULTIPLE**: distancia tipo Jaccard (1 – intersección/unión).
     - **LIBRE**: se usa distancia de **Levenshtein** para comparar textos y se normaliza por la longitud de las cadenas.

   - Para cada otro usuario j, se guarda un objeto VecinoDistancia con el índice del vecino, distancia al usuario actual i el valor que ese vecino tiene para la pregunta que estamos imputando.

   - Se ordenan estos vecinos por distancia y se escogen los K vecinos más cercanos (*K_NEIGHBORS*, fijado a 5 en nuestro código).

3. **Cálculo del valor imputado según el tipo de pregunta**
   Una vez tenemos los K vecinos, el valor a imputar depende del tipo de pregunta (*calcularValorImputado*):

   - **NUMÉRICA**
     Se hace una media ponderada por distancia:

   - **ORDINAL / CATEGORÍA_SIMPLE / LIBRE**
     Se aplica una **mayoría**: se cuentan las ocurrencias de cada respuesta y se escoge la más frecuente.

   - **CATEGORÍA_MULTIPLE**
     Se cuentan cuántas veces aparece cada opción entre los vecinos y se seleccionan aquellas opciones que aparecen más de la mitad de las veces (mayoría estricta), generando un conjunto final de opciones.

   - Si por algún motivo no hay suficiente información fiable, se recurre a un **valor por defecto** calculado por *obtenerValorPorDefecto(...),* por ejemplo con la media del rango para numéricas, opción central para ordinales, cadena vacía o conjunto vacío cuando no hay respuesta razonable.

Gracias a esto, conseguimos:

- Eliminar todos los null antes de pasar los datos al algoritmo de clustering.
- Aprovechar la estructura de los datos: cada imputación se hace usando usuarios "parecidos" según el resto de respuestas.
- Soportar todos los tipos de pregunta de la encuesta sin romper la ejecución del clustering.

Elbow method

Al ejecutar el clustering, nuestro codigo de CtrlDominio hace uso de la técnica del codo. Esta técnica trata de ejecutar el clustering para distintas k (hasta el límite deseado) y usar finalmente la que mayor precisión tenga. Puesto que no controlamos donde empezará el primer cluster, repetimos varias veces el algoritmo para cada k y nos quedamos con la iteración que tenga mejor resultado. Esto se hace en el metodo *ejecutarClustering(...)* de CtrlDominio y se prueba durante los test. Estos valores de calidad se obtienen directamente de los campos *silhouette* e *inercia* del *ResultadoClustering* devuelto por cada ejecución, de manera que el controlador no necesita recalcular nada, solo comparar resultados.

La idea es:

1. **Explorar distintos valores de k**
   Para una encuesta dada, se ejecuta el algoritmo de clustering (K-Means, K-Means++ o K-Medoids) para distintos valores de k dentro de un rango, por ejemplo k = 2, 3, ..., *k_max*.

2. **Múltiples ejecuciones por cada k**
   Los algoritmos basados en inicialización aleatoria pueden caer en **mínimos locales**: con una mala inicialización pueden dar un clustering de peor calidad.
   Para mitigar esto:

   - Para cada valor de k se lanzan varias ejecuciones del algoritmo con distintas semillas aleatorias.

   - De esas ejecuciones, nos quedamos con la que tenga **mejor calidad**, medida por:

     - **Silhouette** (cohesión/separación),
     - I**nercia** (compactación interna).

3. **Cálculo de la "curva del codo"**
   Para cada k se obtiene un valor de calidad (por ejemplo silhouette media o inercia total). Conceptualmente, si dibujásemos:

   - k en el eje X y la métrica (silhouette o inercia) en el eje Y
     veríamos una curva que, a partir de cierto k, mejora mucho menos: ahí es donde se produce el "codo".

4. **Selección de k**
   El controlador se queda con el valor de k que ofrece el mejor compromiso entre calidad alta del clustering y no disparar el número de clusters.

En nuestro caso, esta lógica está encapsulada en el controlador de dominio (*CtrlDominio*), que prepara los datos, ejecuta múltiples veces el algoritmo para cada k y finalmente devuelve el *ResultadoClustering* correspondiente a la mejor combinación algoritmo+k+inicialización.

Inercia

Además del coeficiente de silhouette nuestro algoritmo también calcula la inercia total del resultado que otorga cada ejecución de clustering. Se calcula en el método *calcularInercia()* de su algoritmo respectivo. Este valor de inercia se almacena en el campo *inercia* de *ResultadoClustering* (véase el apartado **4.2**) y se utiliza tanto para evaluar la calidad interna del clustering como, opcionalmente, como métrica para construir la curva del codo cuando se prueban distintos valores de k.

La inercia se define como la suma, para todos los puntos, de la distancia al centro de su cluster al cuadrado:

$$\text{Inercia} = \sum_{i=1}^{n} d(x_i, c_{g(i)})^2$$

donde:

- x_i es el usuario i
- $c_{g(i)}$ es el centro (o medoide) del cluster al que pertenece
- d() es la función de distancia que ya usamos en el algoritmo

5. Jocs de proves

5.1. Introducción

Este documento describe de forma **exhaustiva** las pruebas realizadas sobre el **ejecutable del proyecto** correspondiente a la tercera entrega
El plan se ha diseñado **siguiendo explícitamente la Guía de Datos de Prueba oficial**, que proporciona usuarios, encuestas y estructura de datos precargados para facilitar la validación del sistema y la corrección interactiva

Las pruebas cubren los principales **casos de uso del sistema**, verifican la correcta persistencia de la información y evalúan la robustez del programa ante situaciones normales y erróneas.

Comandos Útiles

Shell

```
# Ejecutar aplicación con datos precargados
./run.sh

# Limpiar solo datos (mantiene código compilado)
make clean-data

# Limpiar todo y recompilar
./run.sh --rebuild

# Ejecutar tests
./run.sh --test

# Ver estructura de datos
tree data/

# Regenerar Javadoc
make javadoc
```

5.2. Entorno de pruebas

- **Lenguaje / Plataforma:** *Java 21*
- **Ejecutable:** *EXE/aplicacion.jar*
- **Directorio de ejecución:** raíz del proyecto

Comando de ejecución:
java -jar EXE/aplicacion.jar

5.3. Datos de prueba utilizados

De acuerdo con la Guía de Datos de Prueba, el sistema dispone de datos precargados en la capa de persistencia

5.3.1 Usuarios precargados

Administradores

- *admin / admin* – Administrador principal
- *admin2 / admin123* – Administrador secundario
- *supervisor / super* – Usuario supervisor

Permisos: creación, modificación y eliminación de encuestas, gestión de preguntas, visualización de resultados y ejecución de clustering.

Respondedores

- *user001 … user010* (contraseña común: *user123*)

Permisos: explorar encuestas, responder encuestas y consultar sus propias respuestas.

5.3.2 Encuestas precargadas

Las siguientes encuestas se utilizan como base para las pruebas:

- **Satisfacción del Cliente (*sat*)**
  Uso: análisis de satisfacción, NPS y validación de preguntas ordinales y numéricas.
- **Hábitos Tecnológicos (*tec*)**
  Uso: clustering por patrones de uso y categorías múltiples.
- **Bienestar y Salud (*sal*)**
  Uso: estadísticas numéricas, correlaciones y clustering.
- **Educación Online (*edu*)**
  Uso: análisis comparativo de ventajas y desafíos.

5.3.3 Ubicación de los ficheros

```
EXE/data/
 ├── usuarios/
 │    ├── admins.json
 │    └── respondedores.json
 ├── encuestas/
 │    ├── sat.json
 │    ├── tec.json
 │    ├── sal.json
 │    └── edu.json
 ├── respuestas/
 │    └── todas_respuestas.csv
 └── clustering/
      ├── clustering_sal.json
      └── clustering_sat.json
```

5.4. Plan de pruebas

PR-01 — Arranque del sistema con datos precargados

- **Objeto de la prueba:** Verificar que la aplicación arranca correctamente cargando los datos de ejemplo.
- **Ficheros de datos necesarios:**
  EXE/data/usuarios/.json,*
  EXE/data/encuestas/.json.*
- **Valores estudiados:** Conjunto completo de datos precargados (caja negra).
- **Efectos estudiados:** Aparición del menú inicial sin errores.
- **Resultado esperado:** El sistema se inicia correctamente y muestra el menú principal.

PR-02 — Login como administrador

- **Objeto de la prueba:** Caso de uso “Iniciar sesión como administrador”.
- **Ficheros de datos necesarios:** *EXE/data/usuarios/admins.json.*
- **Valores estudiados:** Credenciales válidas *(admin / admin).*
- **Efectos estudiados:** Navegación al menú de administración.
- **Resultado esperado:** Acceso correcto al menú de administrador.

PR-03 — Login inválido

- **Objeto de la prueba:** Validación de credenciales incorrectas.
- **Ficheros de datos necesarios:** *EXE/data/usuarios/*.json.*
- **Valores estudiados:** Usuario inexistente o contraseña errónea.
- **Efectos estudiados:** Mensaje de error y permanencia en la vista de login.
- **Resultado esperado:** No se inicia sesión.

PR-04 — Login como respondedor

- **Objeto de la prueba:** Caso de uso “Iniciar sesión como respondedor”.
- **Ficheros de datos necesarios:** *EXE/data/usuarios/respondedores.json*.
- **Valores estudiados:** *user001 / user123*.
- **Efectos estudiados:** Acceso al menú de respondedor.
- **Resultado esperado:** Sesión iniciada correctamente.

PR-05 — Exploración de encuestas

- **Objeto de la prueba:** Caso de uso “Explorar encuestas”.
- **Ficheros de datos necesarios:** *EXE/data/encuestas/*.json*.
- **Valores estudiados:** Conjunto completo de encuestas precargadas.
- **Efectos estudiados:** Visualización de listas, selección y navegación.
- **Resultado esperado:** Se muestran todas las encuestas disponibles.

PR-06 — Responder encuesta como usuario

- **Objeto de la prueba:** Caso de uso “Responder encuesta”.
- **Ficheros de datos necesarios:**
  EXE/data/encuestas/sat.json,
  EXE/data/respuestas/todas_respuestas.csv.
- **Valores estudiados:** Respuestas válidas a todas las preguntas obligatorias.
- **Efectos estudiados:** Progreso por pantalla, selección de opciones y confirmación.
- **Resultado esperado:** Respuestas guardadas correctamente.

PR-07 — Validación de preguntas obligatorias

- **Objeto de la prueba:** Verificar que no se permite finalizar sin responder preguntas obligatorias.
- **Ficheros de datos necesarios:** mismos que PR-06.
- **Valores estudiados:** Campos obligatorios sin responder.
- **Efectos estudiados:** Mensaje de error y bloqueo del envío.
- **Resultado esperado:** No se registran respuestas incompletas.

PR-08 — Visualización de resultados (Administrador)

- **Objeto de la prueba:** Caso de uso “Ver resultados”.
- **Ficheros de datos necesarios:** *EXE/data/respuestas/todas_respuestas.csv*.
- **Valores estudiados:** Respuestas reales de usuarios.
- **Efectos estudiados:** Visualización de tablas y estadísticas.
- **Resultado esperado:** Resultados coherentes y sin errores.

PR-09 — Ejecución de clustering

- **Objeto de la prueba:** Caso de uso “Ejecutar clustering”.
- **Ficheros de datos necesarios:**
  EXE/data/encuestas/sal.json
  EXE/data/respuestas/todas_respuestas.csv
  EXE/data/clustering/
- **Valores estudiados:** Diferentes valores de K (2–5).
- **Efectos estudiados:** Visualización de grupos y manejo de errores.
- **Resultado esperado:** Clustering ejecutado correctamente.

5.5. Conclusión

El presente plan de pruebas valida exhaustivamente el ejecutable del proyecto utilizando los datasets que proporcionamos en nuestra carpeta *data_seed*.
Las pruebas cubren todos los casos de uso principales, verifican la persistencia de datos y garantizan la robustez del sistema, cumpliendo plenamente con los requisitos de la tercera entrega.