

UNIVERSITAT POLITÈCNICA  
DE CATALUNYA  
BARCELONATECH

---

## 2<sup>a</sup> ENTREGA PROP

---

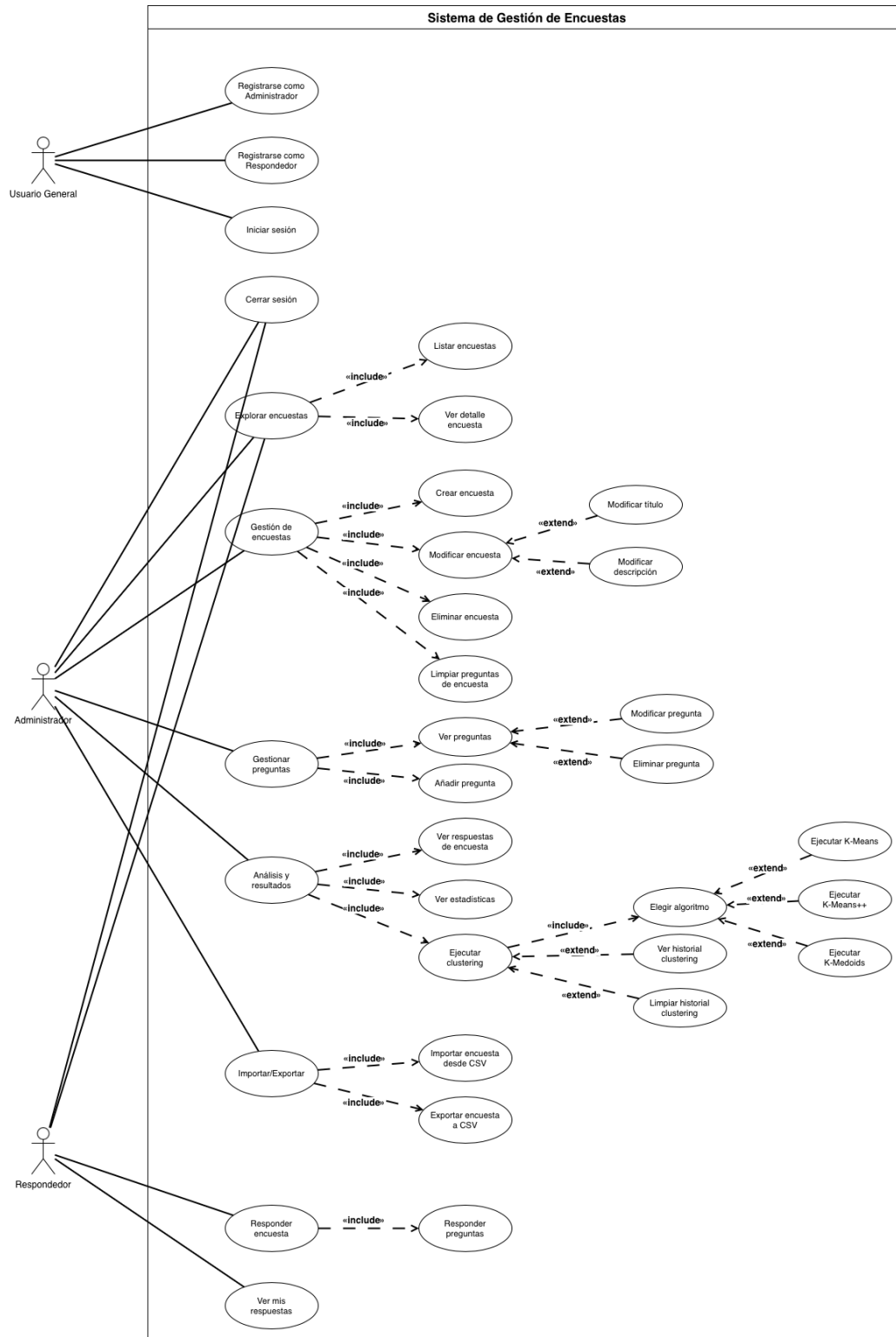
Identificador de l'equip: 41.1

Mohamed Amara El Houti :	<a href="mailto:mohamed.amara.el.houti@estudiantat.upc.edu">mohamed.amara.el.houti@estudiantat.upc.edu</a>
Ossama Chaer Dalerou :	<a href="mailto:ossama.chaer@estudiantat.upc.edu">ossama.chaer@estudiantat.upc.edu</a>
Arnau Serra Florenciano :	<a href="mailto:arnau.serra.florenciano@estudiantat.upc.edu">arnau.serra.florenciano@estudiantat.upc.edu</a>
Adam Ziani Hassun :	<a href="mailto:adam.ziani@estudiantat.upc.edu">adam.ziani@estudiantat.upc.edu</a>

# Índice

<b>1. Casos de Uso.....</b>	<b>1</b>
<b>2. Diagrama de clases del modelo conceptual.....</b>	<b>11</b>
2.1 Visión general del diagrama de clases.....	12
2.2 Descripción de las clases de dominio.....	13
2.3 Descripción de las clases de servicio y control.....	19
<b>3. Diagramas de las capas y su descripción.....</b>	<b>23</b>
3.1. Capa de dominio.....	23
3.2. Capa de Persistencia.....	48
3.3. Capa de Presentación.....	52
<b>4. Estructuras de datos y algoritmos utilizados.....</b>	<b>66</b>
4.1 Estructuras de datos.....	66
4.2 Algoritmos.....	68
<b>5. Algoritmos de Clustering.....</b>	<b>74</b>
5.1 Idea general.....	74
5.2 ResultadoClustering.....	74
5.3 AlgoritmoClustering.....	75
5.4 KMeans.....	76
5.5 KMeansPlusPlus.....	77
5.6 KMedoids.....	78
5.7 Funcionalidades extras.....	79
<b>6. Jocs de proves.....</b>	<b>83</b>
6.1 TestGestorEncuestas.....	83
6.2. TestGestorUsuarios.....	86
6.3. TestGestorRespuestas.....	89
6.4. TestGestorClustering.....	92
6.5 Resumen de cobertura.....	98

# 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. Diagrama de clases del modelo conceptual

---



**IMPORTANTE: ESTA ES LA PRIMERA VERSIÓN DEL DIAGRAMA, LEER EL README PARA MÁS INFORMACIÓN!!**

## 2.1 Visión general del diagrama de clases

El sistema modela una aplicación de gestión de encuestas y clustering de respuestas. El modelo se organiza en tres bloques principales:

### 1. Bloque de dominio de encuestas

- Clases: *Encuesta*, *Pregunta* y sus subclases (*Numerica*, *Ordinal*, *CategoriaSimple*, *CategoriaMultiple*, *Libre*), *Respuesta*, *Usuario* y sus subclases (*UsuarioRespondedor*, *UsuarioAdmin*), el *enum TipoPregunta* y la clase de resultado *ResultadoClustering*.
- Aquí está toda la lógica de cómo funciona nuestra gestión: qué es una encuesta, qué tipos de preguntas hay, cómo se valida una respuesta y cómo se representan los resultados del *clustering*.

### 2. Bloque de servicios de entrada/salida

- Clases: *LectorCSV*, *EscritorCSV* y el controlador *CtrlCSV*.
- Se encargan de importar encuestas y respuestas desde **CSV** y de exportar los resultados a **CSV**, aislando los detalles de ficheros del resto del dominio.

### 3. Bloque de control y clustering

- Clases: *GestorEncuestas*, *GestorUsuarios*, *GestorRespuestas*, *GestorClustering*, el contexto *Clustering*, las estrategias *KMeans*, *KMedoids*, *KMeansPlusPlus* y el *CtrlDominio*.
- Este bloque relaciona los objetos de dominio, aplica los algoritmos de clustering y proporciona una interfaz básica para la futura **capa de presentación**.

Identificadores y asociaciones clave:

- Cada *Encuesta* tiene un *id* : *String* único.
- Cada *Pregunta* tiene un *id* : *String* único y pertenece exactamente a **una** *Encuesta*.
- Cada *Usuario* (tanto admin como respondedor) tiene un *id* : *String* único.
- *Respuesta* se identifica por el triplete (*idUsuario*, *idPregunta*, *idEncuesta*), almacenado como tres atributos.
- La relación encuesta–pregunta es de composición: una *Encuesta* contiene 1..\* *Pregunta*.
- Un *Usuario* puede responder 0..\* encuestas, y una *Encuesta* puede tener respuestas de 0..\* usuarios. Las respuestas de un usuario se agrupan por encuesta.

## 2.2 Descripción de las clases de dominio

### 2.2.1 Clase Encuesta

- **Nombre:** Encuesta (*main.domain.classes.Encuesta*)
- **Descripción breve:** representa una encuesta con un identificador único, un título, una descripción y una lista ordenada de preguntas.
- **Atributos principales:**
  - *id* : *String* – identificador único (*UUID*).
  - *titulo* : *String* – título de la encuesta.
  - *descripcion* : *String* – descripción corta de la encuesta.
  - *preguntas* : *List<Pregunta>* – lista ordenada de preguntas que componen la encuesta.
- **Relaciones:**
  - Una *Encuesta* contiene 1..\* *Pregunta*.
  - Es usada por *GestorEncuestas*, *CtrlDominio*, *GestorRespuestas* y *GestorClustering*.
- **Métodos principales:**
  - Constructora: *Encuesta(String titulo, String descripcion)*
  - Consultoras: *getId()*, *getTitulo()*, *getDescripcion()*, *getPreguntas()*, *getNumPreguntas()*, *getPregunta(int)*
  - Modificadoras: *setTitulo(String)*, *setDescripcion(String)*, *agregarPregunta(Pregunta)*, *modificarPregunta(Pregunta, int)*, *eliminarPregunta(int)*, *eliminarTodasPreguntas()*, *getIndicePregunta(Pregunta)*

### 2.2.2 Clase abstracta Pregunta y enum TipoPregunta

- **Nombre:** Pregunta (*main.domain.classes.Pregunta*)
- **Descripción breve:** clase abstracta que agrupa lo común a todas las preguntas: identificador, enunciado, obligatoriedad y la capacidad de validar una respuesta.
- **Atributos:**
  - *id* : *String* – identificador único (*UUID*).
  - *enunciado* : *String* – texto de la pregunta.
  - *obligatoria* : *boolean* – indica si la pregunta debe contestarse obligatoriamente.
- **Métodos principales:**
  - Constructoras protegidas:
    - *Pregunta(String enunciado)*
    - *Pregunta(String enunciado, boolean obligatoria)*
  - Consultoras: *getId()*, *getEnunciado()*, *esObligatoria()*
  - Modificadoras: *setEnunciado(String)*, *setObligatoria(boolean)*
  - Abstractas:
    - *getTipoPregunta() : TipoPregunta*
    - *validarRespuesta(Object respuesta) : boolean* – comprueba que el valor sea válido para este tipo de pregunta.

- **Enum** *TipoPregunta* (*main.domain.classes.TipoPregunta*):
  - Valores: NUMERICA, ORDINAL, CATEGORIA\_SIMPLE, CATEGORIA\_MULTIPLE, LIBRE.
  - Se usa tanto en las subclases de *Pregunta* como en los algoritmos de *clustering* para decidir el tratamiento de cada columna.

### 2.2.3 Subclase Numerica

- **Nombre:** *Numerica* (*main.domain.classes.Numerica*)
- **Descripción breve:** representa una pregunta **numérica** (por ejemplo, edad, nota, etc.) con un rango opcional [min, max].
- **Atributos adicionales:**
  - *min* : *Double* – valor mínimo permitido (puede ser *null* si no hay límite).
  - *max* : *Double* – valor máximo permitido (puede ser *null*).
- **Relaciones:**
  - Hereda de *Pregunta*.
  - El rango se usa en el *clustering* para normalizar las distancias numéricas.
- **Métodos principales:**
  - Constructoras:
    - *Numerica(String enunciado, Double min, Double max)*
    - *Numerica(String enunciado)* (sin restricciones explícitas)
  - Consultoras: *getMin()*, *getMax()*
  - Modificadora: *setRango(Double min, Double max)* (valida que  $\text{min} \leq \text{max}$  cuando ambos no son nulos).
  - Overrides:
    - *getTipoPregunta()* → *TipoPregunta.NUMERICA*
    - *validarRespuesta(Object valor)* – convierte a *Double* (desde *Number* o *String*), rechaza *NaN*/infinito y comprueba que el valor respete el rango si hay min/max.

### 2.2.4 Subclase Ordinal

- **Nombre:** *Ordinal* (*main.domain.classes.Ordinal*)
- **Descripción breve:** pregunta donde la respuesta es una opción textual de un conjunto ordenado (por ejemplo, “Bajo”, “Medio”, “Alto”).
- **Atributos adicionales:**
  - *opciones* : *Set<String>* – conjunto de opciones ordinales válidas.
- **Relaciones:**
  - Hereda de *Pregunta*.
  - Sus opciones se usan en el *clustering* para definir distancias **ordinales**.
- **Métodos principales:**
  - Constructora: *Ordinal(String texto, Set<String> opciones)*
  - Consultora: *getOpciones()* : *Set<String>* (devuelve una copia defensiva).
  - Overrides:
    - *getTipoPregunta()* → *TipoPregunta.ORDINAL*
    - *validarRespuesta(Object respuesta)* – acepta *String* no vacía incluida en opciones (o *null* si la pregunta no es obligatoria).



### 2.2.5 Subclases CategoriaSimple y CategoriaMultiple

CategoriaSimple:

- **Nombre:** CategoriaSimple (*main.domain.classes.CategoriaSimple*)
- **Descripción breve:** pregunta de **selección única** dentro de un conjunto **finito** de opciones (por ejemplo, “Color preferido”).
- **Atributos:**
  - *opciones* : *Set<String>* – conjunto de opciones disponibles.
- **Métodos principales:**
  - Constructora: *CategoriaSimple(String enunciado, Set<String> opciones)*
  - Consultoras: *getOpciones()*
  - Modificadoras de conveniencia: *agregarOpciones(String)*, *eliminarOpcion(String)*
  - Overrides:
    - *getTipoPregunta()* → *TipoPregunta.CATEGORIA\_SIMPLE*
    - *validarRespuesta(Object respuesta)* – pide un String que sea una de las opciones (o vacío/null si no es obligatoria).

CategoriaMultiple

- **Nombre:** CategoriaMultiple (*main.domain.classes.CategoriaMultiple*)
- **Descripción breve:** pregunta que permite seleccionar varias opciones de un conjunto, con límite máximo de selecciones.
- **Atributos:**
  - *opciones* : *Set<String>* – conjunto de opciones posibles.
  - *maxSelecciones* : *int* – número máximo de opciones seleccionables.
- **Métodos principales:**
  - Constructora: *CategoriaMultiple(String enunciado, Set<String> opciones, int maxSelecciones)*
  - Consultoras: *getOpciones()*, *getMaxSelecciones()*
  - Overrides:
    - *getTipoPregunta()* → *TipoPregunta.CATEGORIA\_MULTIPLE*
    - *validarRespuesta(Object respuesta)* – espera un *Set<?>* de cadenas, subconjunto de opciones y con tamaño  $\leq$  maxSelecciones.

### 2.2.6 Subclase Libre

- **Nombre:** Libre (*main.domain.classes.Libre*)
- **Descripción breve:** modela una pregunta de respuesta abierta (**texto libre**), con una **longitud máxima** configurable.
- **Atributos:**
  - *longitudMaxima* : *int* – máximo de caracteres permitidos (por defecto 1000).
- **Métodos principales:**
  - Constructoras: *Libre(String enunciado)*, *Libre(String enunciado, int longitudMaxima)*
  - Consultora: *getLongitudMaxima()*

- Modificadora: *setLongitudMaxima(int)*
- Overrides:
  - *getTipoPregunta() → TipoPregunta.LIBRE*
  - *validarRespuesta(Object valor)* – permite *null* si no es obligatoria; si hay texto, comprueba que la longitud no supere *longitudMaxima*.

### 2.2.7 Clase Respuesta

- **Nombre:** *Respuesta* (*main.domain.classes.Respuesta*)
- **Descripción breve:** representa la respuesta de un usuario a una pregunta concreta de una encuesta.
- **Atributos:**
  - *idUsuario : String*
  - *idPregunta : String*
  - *idEncuesta : String*
  - *valor : Object* – valor de la respuesta (puede ser numérico, texto, conjunto de opciones, etc.).
  - *contestada : boolean* – indica si la pregunta está contestada o no.
- **Métodos principales:**
  - Constructoras:
    - *Respuesta(String idUsuario, String idPregunta, String idEncuesta, Object valor)*
    - *Respuesta(String idUsuario, String idPregunta, String idEncuesta)* (sin valor inicial)
  - Consultoras: *getIdUsuario(), getIdPregunta(), getIdEncuesta(), getValor(), estaContestada()*
  - Modificadoras: *setValor(Object), limpiar()* (pone el valor a *null* y marca *contestada = false*)
  - Redefinición de *equals()* y *hashCode()* basada en (*idUsuario, idPregunta, idEncuesta*).

### 2.2.8 Clase abstracta Usuario y subclases

#### Usuario

- **Nombre:** *Usuario* (*main.domain.classes.Usuario*)
- **Descripción breve:** superclase común para cualquier usuario del sistema (administrador o respondedor).
- **Atributos:**
  - *id : String*
  - *nombre : String*
  - *password : String*
- **Métodos:**
  - Consultoras: *getId(), getNombre(), getPassword()*
  - Modificadoras: *setId(String), setNombre(String), setPassword(String)*

- Abstractas:
  - *addRespuesta(String idEncuesta, Respuesta respuesta)*
  - *getRespuestasEncuesta(String idEncuesta) : List<Respuesta>*

## UsuarioRespondedor

- **Nombre:** UsuarioRespondedor (*main.domain.classes.UsuarioRespondedor*)
- **Descripción breve:** usuario que responde encuestas. Almacena sus respuestas agrupadas por encuesta.
- **Atributos adicionales:**
  - *respuestasPorEncuesta : Map<String, List<Respuesta>>*
- **Métodos relevantes:**
  - Constructoras con y sin password.
  - *addRespuesta(String idEncuesta, Respuesta respuesta)*
  - *getRespuestasEncuesta(String idEncuesta)*
  - *haRespondidoEncuesta(String idEncuesta) : boolean*
  - *getNumeroEncuestasRespondidas() : int*
  - *eliminarRespuestasEncuesta(String idEncuesta)*

## UsuarioAdmin

- **Nombre:** UsuarioAdmin (*main.domain.classes.UsuarioAdmin*)
- **Descripción breve:** usuario con permisos de administración (crear/modificar encuestas) que también puede responder encuestas.
- **Atributos adicionales:**
  - *respuestasPorEncuesta : Map<String, List<Respuesta>>* (mismo patrón que el respondedor).
- **Métodos relevantes:**
  - Constructoras con y sin password.
  - *addRespuesta(String idEncuesta, Respuesta respuesta)*
  - *getRespuestasEncuesta(String idEncuesta)*
  - *haRespondidoEncuesta(String idEncuesta) : boolean*

## 2.2.9 Clase ResultadoClustering

- **Nombre:** ResultadoClustering (*main.domain.classes.ResultadoClustering*)
- **Descripción breve:** encapsula el resultado de una ejecución de clustering.
- **Atributos:**
  - *groups : int[]* – para cada usuario, el índice de cluster asignado.
  - *centers : Object[][]* – centros (o medoides) de cada cluster.
  - *silhouette : double* – índice de silueta global.
  - *algoritmo : String* – nombre del algoritmo empleado.
  - *k : int* – número de clusters.
  - *numIteraciones : int* – número de iteraciones realizadas.
  - *idsUsuarios : List<String>* – lista de IDs de usuario en el mismo orden que groups.

- **Métodos principales:**
  - Constructor básico y constructor enriquecido (con metadatos).
  - Consultoras: *getGroups()*, *getCenters()*, *getSilhouette()*, *getAlgoritmo()*, *getK()*, *getNumIteraciones()*
  - *setIdsUsuarios(List<String>)*
  - *getUsuariosPorGrupo() : List<List<String>>* – devuelve la lista de usuarios agrupados por cluster.
  - *toString()* – resumen legible del resultado.

### 2.2.10 Clases LectorCSV y EscritorCSV

#### LectorCSV

- **Nombre:** *LectorCSV* (*main.domain.classes.LectorCSV*)
- **Descripción breve:** clase de bajo nivel para leer ficheros CSV, con separador configurable y opción de ignorar líneas vacías. Tiene una clase interna *DatosCSV*.
- **Atributos:**
  - *separador : String*
  - *saltarLineasVacias : boolean*
- **Métodos principales:**
  - *leerArchivo(String ruta) : List<String[]>*
  - *leerPrimeraLinea(String ruta) : String[]*
  - *leerSinPrimeraLinea(String ruta) : List<String[]>*
  - *contarLineas(String ruta) : int*
  - *contarColumnas(String ruta) : int*
  - *leerConEncabezados(String ruta) : DatosCSV*
  - Getters y setters del separador y de *saltarLineasVacias*.
- **Clase interna DatosCSV:**
  - Atributos: *encabezados : String[], filas : List<String[]>*
  - Métodos: *getEncabezados()*, *getFilas()*, *getNumeroColumnas()*, *getNumeroFilas()*, *getFila(int)*, *getValor(int fila, int columna)*

#### EscritorCSV

- **Nombre:** *EscritorCSV* (*main.domain.classes.EscritorCSV*)
- **Descripción breve:** encapsula la escritura de datos a CSV, con control de separador y encabezados.
- **Atributos:**
  - *separador : String*
  - *incluirEncabezados : boolean*
- **Método principal:**
  - *escribirArchivo(String rutaArchivo, String[] encabezados, List<String[]> filas) : void*

## 2.3 Descripción de las clases de servicio y control

### 2.3.1 Clase GestorEncuestas

- **Nombre:** GestorEncuestas (*main.domain.controller.GestorEncuestas*)
- **Descripción breve:** gestor *CRUD* de encuestas. Mantiene todas las encuestas en memoria y centraliza su creación, modificación y borrado.
- **Atributos:**
  - *encuestas* : *Map<String, Encuesta>* – diccionario *idEncuesta* → *Encuesta*.
- **Métodos principales:**
  - Crear/añadir: *crearEncuesta(String titulo, String descripcion), addEncuesta(Encuesta)*
  - Consultar: *obtenerEncuesta(String id), listarEncuestas(), existeEncuesta(String id), getNumeroEncuestas()*
  - Modificar: *modificarEncuesta(String id, String nuevoTitulo, String nuevaDescripcion)*
  - Eliminar: *eliminarEncuesta(String id)*
  - Gestión de preguntas: *addPregunta(String idEncuesta, Pregunta), modificarPregunta(String idEncuesta, int index, Pregunta), eliminarPregunta(String idEncuesta, int index), eliminarTodasPreguntas(String idEncuesta)*

### 2.3.2 Clase GestorUsuarios

- **Nombre:** GestorUsuarios (*main.domain.controller.GestorUsuarios*)
- **Descripción breve:** gestiona la creación y consulta de usuarios, distinguiendo entre respondedores y administradores.
- **Atributos:**
  - *usuariosRespondedores* : *Map<String, UsuarioRespondedor>*
  - *usuariosAdmin* : *Map<String, UsuarioAdmin>*
- **Métodos principales:**
  - Creación: *crearUsuarioRespondedor(String id, String nombre, String password)*, versión sin password, y *crearUsuarioAdmin(String id, String nombre, String password)*
  - Consulta: *obtenerUsuario(String id), esAdmin(String id), existeUsuario(String id), listarUsuarios()*
  - Respuestas: *obtenerRespuestasUsuario(String idUsuario, String idEncuesta) : List<Respuesta>*
  - Análisis: *obtenerUsuariosQueRespondieron(String idEncuesta) : List<Usuario>*

### 2.3.3 Clase GestorRespuestas

- **Nombre:** GestorRespuestas (*main.domain.controller.GestorRespuestas*)
- **Descripción breve:** servicio encargado de validar y registrar respuestas de usuarios a encuestas. No mantiene estado propio.

- **Métodos principales:**

- *responderPregunta(Usuario usuario, Encuesta encuesta, String idPregunta, Object valor) : Respuesta*
  - Localiza la Pregunta, llama a *validarRespuesta*, y si es válida crea una Respuesta y la añade al Usuario.
- *responderEncuesta(Usuario usuario, Encuesta encuesta, Map<String,Object> respuestasPorPregunta) : void*
  - Verifica que se hayan contestado todas las preguntas obligatorias y llama internamente a *responderPregunta* para cada entrada.
- Métodos de ayuda:
  - *obtenerIndicePregunta(Encuesta encuesta, String idPregunta) : int*
  - *encontrarPregunta(Encuesta encuesta, String idPregunta)* (privado)

### 2.3.4 Clase GestorClustering y contexto Clustering

#### GestorClustering

- **Nombre:** GestorClustering (*main.domain.controller.GestorClustering*)
- **Descripción breve:** prepara los datos para el clustering y configura el algoritmo. Convierte las respuestas en una matriz *Object[][]* y llama al contexto *Clustering*.
- **Atributos:**
  - *idsUsuarios : List<String>* – IDs de los usuarios en el mismo orden que las filas de datos.
- **Métodos principales:**
  - *ejecutarClustering(Clustering clustering, List<Usuario> usuarios, Encuesta encuesta, GestorRespuestas gestorRespuestas) : ResultadoClustering*
  - Métodos privados:
    - *prepararDatos(...)* – construye la matriz de respuestas, sustituyendo valores nulos haciendo uso del algoritmo KNN.
    - *convertirRespuestasAArray(...)*
    - *configurarAlgoritmo(Clustering clustering, Encuesta encuesta)* – pasa rangos numéricos y opciones ordinales.
    - *extraerTiposPreguntas(Encuesta encuesta) : TipoPregunta[]*
    - *imputarValoresNull(List<Object[]> datos, Encuesta encuesta)*
    - *imputarConKNN(...)* - imputa la nueva información a la lista de respuestas otorgada (no modifica el valor real de la respuesta)
    - *calcularDistanciaParcial(...) : Double*
    - *calcularDistanciaUnaPregunta(...) : Double*
    - *calcularValorImputado(...)*
    - *obtenerValorPorDefecto(...)*
    - *levenshteinDistance(...) : Int*
    - *convertToSet(Object obj) : Set<String>*

## Clustering

- **Nombre:** Clustering (*main.domain.classes.Clustering*)
- **Descripción breve:** contexto del patrón *Strategy*. Contiene una referencia a una implementación de *AlgoritmoClustering* y ofrece una interfaz básica para ejecutar el algoritmo y configurar parámetros.
- **Atributos:**
  - *estrategia* : *AlgoritmoClustering*
- **Métodos principales:**
  - *Constructor*: *Clustering(AlgoritmoClustering estrategia)*
  - *setEstrategia(AlgoritmoClustering estrategia)*
  - *ejecutar(Object[][] datos) : ResultadoClustering*
  - *setTiposPreguntas(TipoPregunta[] tipos)*
  - *configurarRangoNumerico(int indicePregunta, double min, double max)*
  - *configurarOpcionesOrdinales(int indicePregunta, Set<String> opciones)*

### 2.3.5 Estrategias KMeans, KMedoids, KMeansPlusPlus (AlgoritmoClustering)

- **Nombre genérico:** KMeans, KMedoids, KMeansPlusPlus
- **Interfaz:** *AlgoritmoClustering* (*main.domain.classes.AlgoritmoClustering*)
- **Descripción breve:** tres implementaciones distintas de clustering que comparten una interfaz común:
  - Reciben una matriz *Object[][]* de datos ya procesados.
  - Usan *TipoPregunta[]*, rangos numéricos y opciones ordinales para calcular distancias adecuadas a cada tipo de pregunta.
  - Devuelven un *ResultadoClustering* con grupos, centros y silueta.
- **Atributos comunes:**
  - *k* : *int* – número de clusters.
  - *maxIter* : *int* – número máximo de iteraciones.
  - *questionTypes* : *TipoPregunta[]*
  - *minValues, maxValues* : *Map<Integer, Double>* – mínimos y máximos por pregunta numérica.
  - *ordinalOptions* : *Map<Integer, Set<String>>* – opciones por pregunta ordinal.
  - *random* : *Random*
- **Métodos de la interfaz AlgoritmoClustering:**
  - *ResultadoClustering execute(Object[][] data)*
  - *void setTipoPreguntas(TipoPregunta[] tipos)*
  - *void setNumericRange(int questionIndex, double min, double max)*
  - *void setOrdinalOptions(int questionIndex, Set<String> options)*

Cada clase concreta implementa:

- Su estrategia de inicialización de centros/medoides.
- Cálculo de distancias según tipo de pregunta (numérica, ordinal, categórica simple, múltiple, libre).
- Recalculo de centros/medoides.
- Cálculo del índice de silueta.

### 2.3.6 Controlador CtrlCSV

- **Nombre:** CtrlCSV (*main.domain.controller.CtrlCSV*)
- **Descripción breve:** controlador de dominio que encapsula toda la lógica de **importación/exportación** de encuestas y respuestas en formato **CSV**, usando *LectorCSV*, *EscritorCSV*, *GestorEncuestas*, *GestorUsuarios* y *GestorRespuestas*.
- **Atributos:**
  - *gestorUsuarios* : *GestorUsuarios*
  - *gestorEncuestas* : *GestorEncuestas*
  - *gestorRespuestas* : *GestorRespuestas*
  - *lectorCSV* : *LectorCSV*
  - *escritorCSV* : *EscritorCSV*
- **Métodos principales:**
  - Importación:
    - *Encuesta importarCSV(String rutaArchivo)*
    - *Encuesta importarCSV(String rutaArchivo, String titulo, String descripcion)*
  - Exportación:
    - *void exportarEncuesta(String idEncuesta, String rutaArchivo)*
    - *InfoExportacion obtenerInfoExportacion(String idEncuesta)*
  - Información/validación:
    - *InfoCSV obtenerInfoCSV(String rutaArchivo)*
    - *boolean validarCSV(String rutaArchivo)*
  - Métodos privados auxiliares para:
    - Extraer opciones por columna.
    - Crear preguntas a partir de los tipos del CSV.
    - Convertir valores de texto a los tipos esperados.

### 2.3.7 Controlador de alto nivel CtrlDominio

- **Nombre:** CtrlDominio (*main.domain.controller.CtrlDominio*)
- **Descripción breve:** representación de la capa de dominio. Agrupa los gestores y controladores internos y expone métodos que usará la futura capa de presentación (*GUI/CLI*).
- **Atributos:**
  - *gestorEncuestas* : *GestorEncuestas*
  - *gestorUsuarios* : *GestorUsuarios*
  - *gestorRespuestas* : *GestorRespuestas*
  - *gestorClustering* : *GestorClustering*
  - *ctrlCSV* : *CtrlCSV*



- **Responsabilidades principales:**

- Encuestas: *crearEncuesta(...)*, *obtenerEncuesta(...)*, *listarEncuestas()*, *modificarEncuesta(...)*, *eliminarEncuesta(...)*, *addPregunta(...)*, *modificarPregunta(...)*, *eliminarPregunta(...)*.
- Usuarios: *crearUsuarioRespondedor(...)*, *crearUsuarioAdmin(...)*, *obtenerUsuario(...)*.
- Respuestas: *responderPregunta(...)*, *responderEncuesta(...)*, *obtenerRespuestasUsuario(...)*.
- Estadísticas y clustering: *obtenerEstadisticasEncuesta(...)*, *ejecutarClustering(...)* y variantes (*ejecutarClusteringKMeans*, *KMedoids*, *KMeansPlusPlus*).
- CSV: *importarEncuestaDesdeCSV(...)*, *obtenerInfoCSV(...)*, *validarCSV(...)*, *exportarEncuestaCSV(...)*, *obtenerInfoExportacion(...)*.

### 3. Diagramas de las capas y su descripción

---

#### 3.1. Capa de dominio

##### 3.1.1. 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.

### 3.1.2. 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.

### 3.1.3. 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 (idUserario, 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.

### 3.1.4. 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.

### 3.1.5. 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  $\leq 2$  valores únicos), cuenta valores únicos (si  $\leq 10$  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.

### 3.1.6. 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

### 3.1.7. 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*.

### 3.1.8. 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

### 3.1.9. 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.

### 3.1.10. 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.

### 3.1.11. 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)

### 3.1.12. 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()

### 3.1.13. 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  $\text{min} \leq \text{max}$
- **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:  $\text{valor} \geq \text{min}$  y  $\text{valor} \leq \text{max}$ . Rechaza NaN e Infinity.

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

### 3.1.14. 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()*

### 3.1.15. 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)

### 3.1.16. 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.

### 3.1.17. 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.

### 3.1.18. 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.

### 3.1.18 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.



### 3.1.19. 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  $O(n \times m)$  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:  $\text{suma}(\text{valor\_vecino} / \text{distancia\_vecino}) / \text{suma}(1/\text{distancia\_vecino})$
  - 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  $1/\text{distancia}$  (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:  $|\text{valor1} - \text{valor2}| / \text{rango}$
- Normaliza a  $[0,1]$  dividiendo por el rango total
- Ejemplo: Pregunta con min=0, max=100. Respuestas 30 y 80  $\rightarrow |30-80|/100 = 0.5$

**ORDINAL:**

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

**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"  $\rightarrow 0.0$ , "Rojo" vs "Azul"  $\rightarrow 1.0$

**CATEGORIA\_MULTIPLE:**

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

## 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.

### 3.1.20. 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

### 3.1.21. 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:**  $|\text{posA}-\text{posB}|/(n-1)$
- **CATEGORIA\_SIMPLE:** 0 si igual, 1 si diferente
- **CATEGORIA\_MULTIPLE:**  $1 - \text{Jaccard}(A,B)$
- **LIBRE:** Levenshtein normalizada

**3.1.22. AdaptadorPregunta**

**Implementa:** IndicePregunta

**Responsabilidad:** Conectar Clustering con CtrlDominio para obtener índices de preguntas

## 3.2. Capa de Persistencia

### 3.2.1. 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*)

### 3.2.2. PersistenciaEncuestas

**Responsabilidad:** Gestión de archivos JSON individuales de encuestas.

**Atributos:**

- *encuestasJSON*: Map<String, String> (id → JSON)
- *directoríoEncuestas*: 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

### 3.2.3. 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

### 3.2.4. 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

### 3.2.5. 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

### 3.2.6. 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

### 3.2.7. 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 o separada del resto)
  - *DatosCSV:* Contenedor de encabezados y lista de FilaRespuesta con métodos de acceso
- **Robustez:** Manejo de FileNotFoundException, IOException, líneas vacías

### 3.2.8. 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

### 3.2.9. 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 TokenType para Set<String>

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

## 3.3. Capa de Presentación

### 3.3.1. CtrlPresentacion

**Responsabilidad:** Controlador central de la capa de presentación que gestiona la navegación entre 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

### 3.3.2. MenuInicialView

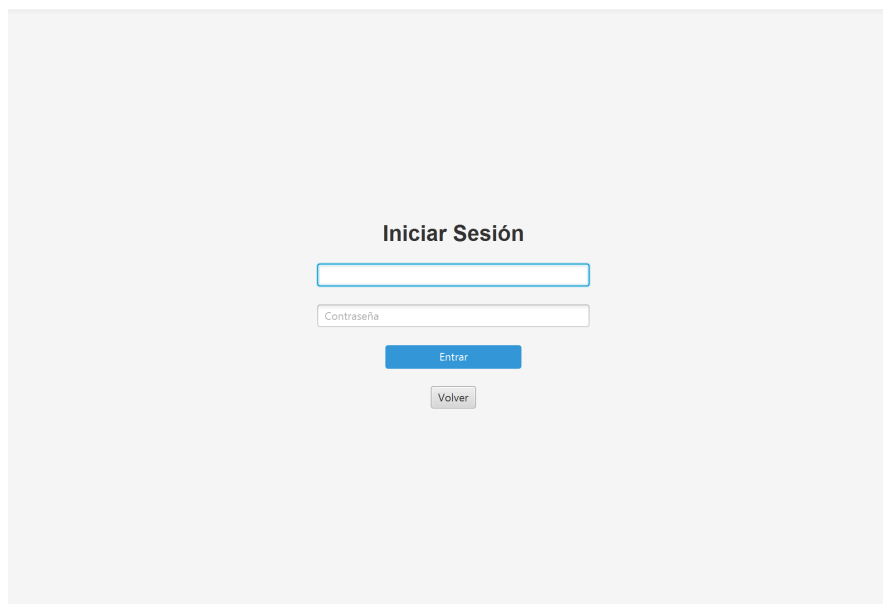


**Responsabilidad:** Pantalla de bienvenida del sistema.

**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

### 3.3.2. LoginView

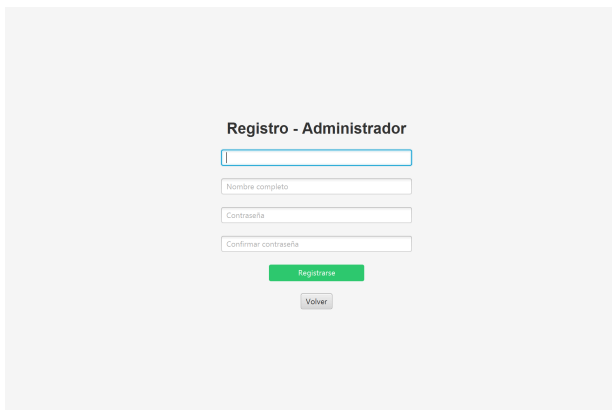


**Responsabilidad:** Pantalla de inicio de sesión.

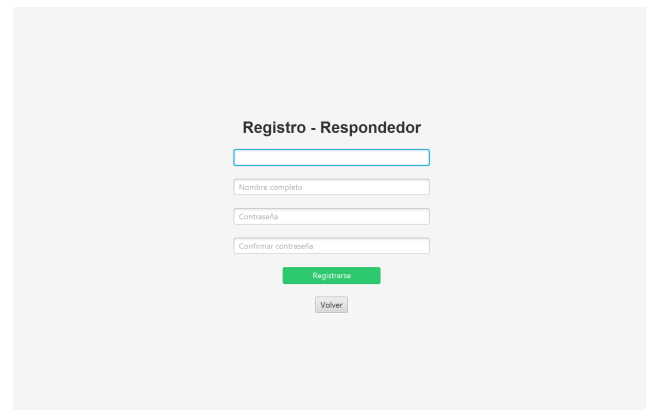
### Funcionalidad:

- 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

### 3.3.3. RegistroView



The screenshot shows a web form titled "Registro - Administrador". It contains a text input field for the username, followed by three input fields for "Nombre completo", "Contraseña", and "Confirmar contraseña". Below these fields are two buttons: a green "Regístrate" button and a grey "Volver" button.



The screenshot shows a web form titled "Registro - Respondedor". It contains a text input field for the username, followed by three input fields for "Nombre completo", "Contraseña", and "Confirmar contraseña". Below these fields are two buttons: a green "Regístrate" button and a grey "Volver" button.

**Responsabilidad:** Formulario de registro de nuevos usuarios.

### Atributos:

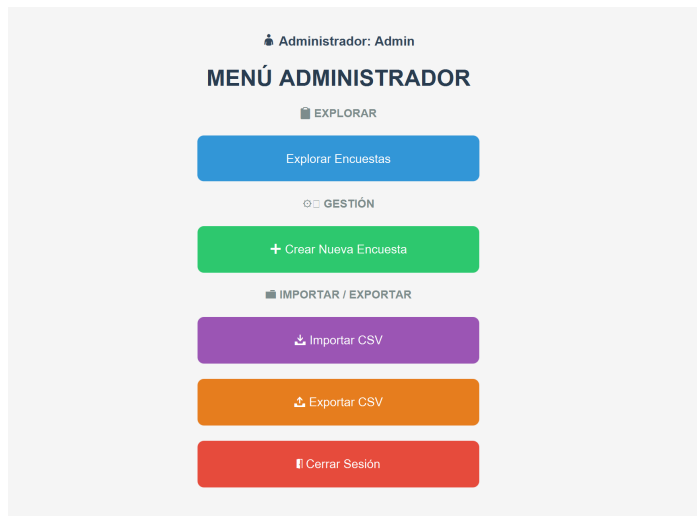
- esAdmin: Flag para determinar tipo de registro

### 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



### 3.3.4. MenuAdminView

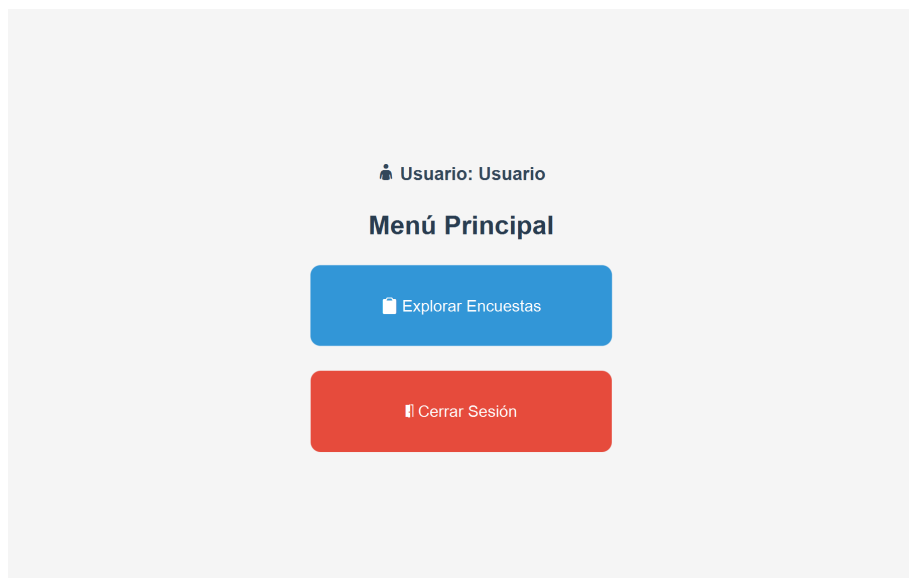


**Responsabilidad:** Menú principal para usuarios administradores.

**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

### 3.3.5. MenuRespondedorView

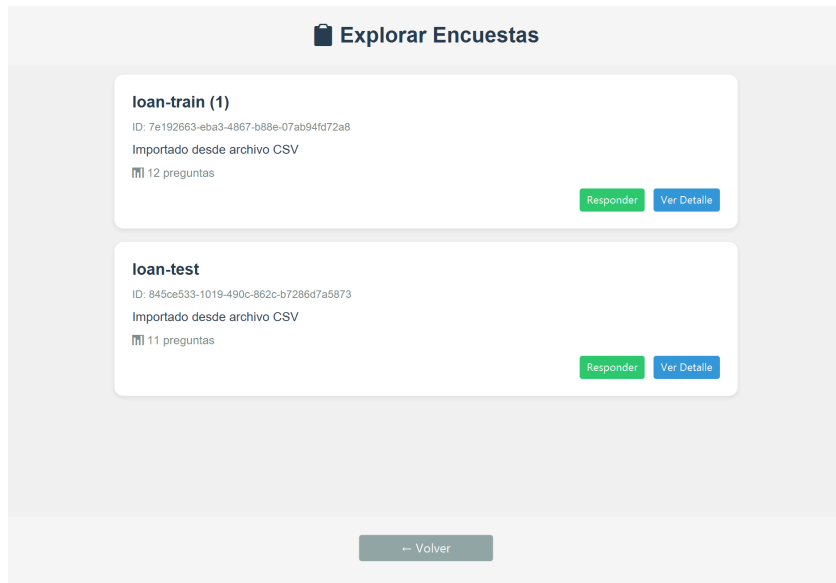


**Responsabilidad:** Menú principal para usuarios respondedores.

### 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

### 3.3.6. 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

### 3.3.7. CrearEncuestaView



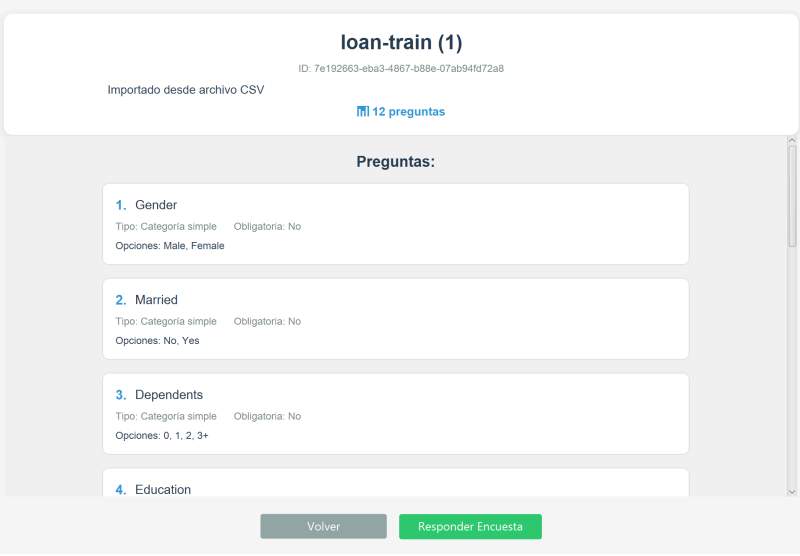
The screenshot shows a web form titled '+ Crear Nueva Encuesta'. It contains two input fields: 'Título de la encuesta:' with a text box, and 'Descripción:' with a larger text area containing the placeholder 'Describe el propósito de la encuesta...'. At the bottom, there are two buttons: a green 'Crear Encuesta' button and a grey 'Cancelar' button.

**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

### 3.3.8. DetalleEncuestaView



The screenshot shows a detailed view of a survey titled 'loan-train (1)' with ID '7e192663-eba3-4867-b88e-07ab94fd72a8'. It indicates the survey was 'Importado desde archivo CSV' and contains '12 preguntas'. Under the 'Preguntas:' section, four questions are listed: 1. Gender (simple category, not mandatory, options: Male, Female), 2. Married (simple category, not mandatory, options: No, Yes), 3. Dependents (simple category, not mandatory, options: 0, 1, 2, 3+), and 4. Education. At the bottom, there are two buttons: a grey 'Volver' button and a green 'Responder Encuesta' button.

**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

### 3.3.9. GestionarEncuestaView

**Gestionar Encuesta**

ID de la Encuesta:  
7e192663-eba3-4867-b88e-07ab94fd72a8

Título:  
loan-train (1)

Descripción:  
Importado desde archivo CSV

**Información**  
Número de preguntas: 12  
Usuarios que respondieron: 614

**Guardar Cambios**

**Volver** **Limpiar Preguntas** **Eliminar Encuesta**

**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

### 3.3.10. GestionarPreguntasView

loan-train (1)

1. Gender	Tipo: Categoría simple	[Eliminar]
2. Married	Tipo: Categoría simple	[Eliminar]
3. Dependents	Tipo: Categoría simple	[Eliminar]
4. Education	Tipo: Categoría simple	[Eliminar]
5. Self_Employed	Tipo: Categoría simple	[Eliminar]
6. ApplicantIncome	Tipo: Numérica	[Eliminar]
7. CoapplicantIncome	Tipo: Numérica	[Eliminar]

+ Añadir Pregunta    ← Volver

**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

### 3.3.11. ResponderEncuestaView



The screenshot shows a web form titled "Responder Encuesta" with a subtitle "loan-train (1)". The form contains four questions, each with a dropdown menu. The first question is "Pregunta 1" with the label "Gender". The second question is "Pregunta 2" with the label "Married". The third question is "Pregunta 3" with the label "Dependents". The fourth question is "Pregunta 4". At the bottom of the form, there are two buttons: "Enviar Respuestas" (green) and "Cancelar" (gray).

**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

### 3.3.12. EstadísticasView



**Responsabilidad:** Visualización de estadísticas de una encuesta.

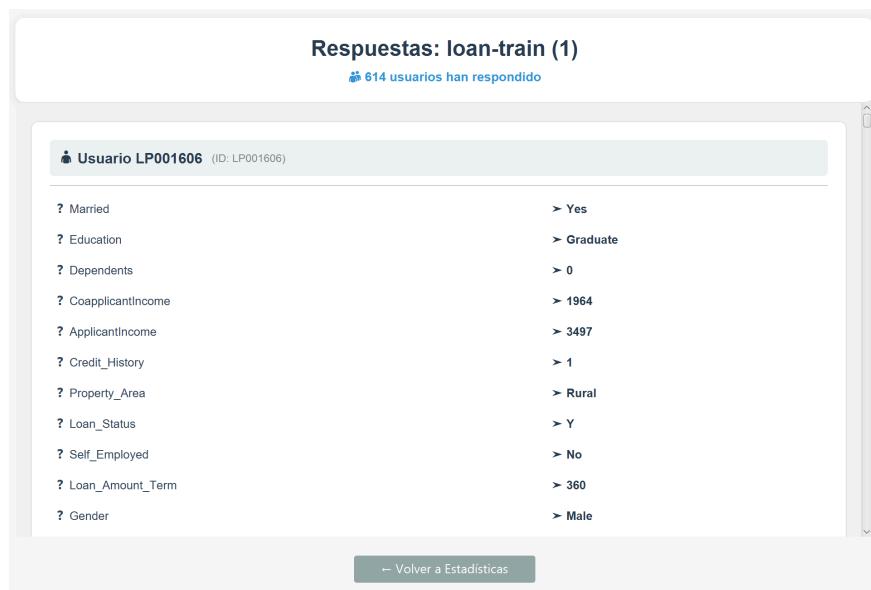
**Atributos:**

- idEncuesta: ID de la encuesta

**Funcionalidad:**

- **Resumen general:** Cards con número de preguntas y respuestas
- **Estadísticas por pregunta:**
  - **Númerica:** 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

### 3.3.13. VerRespuestasView



**Responsabilidad:** Vista detallada de respuestas individuales por usuario.

#### Atributos:

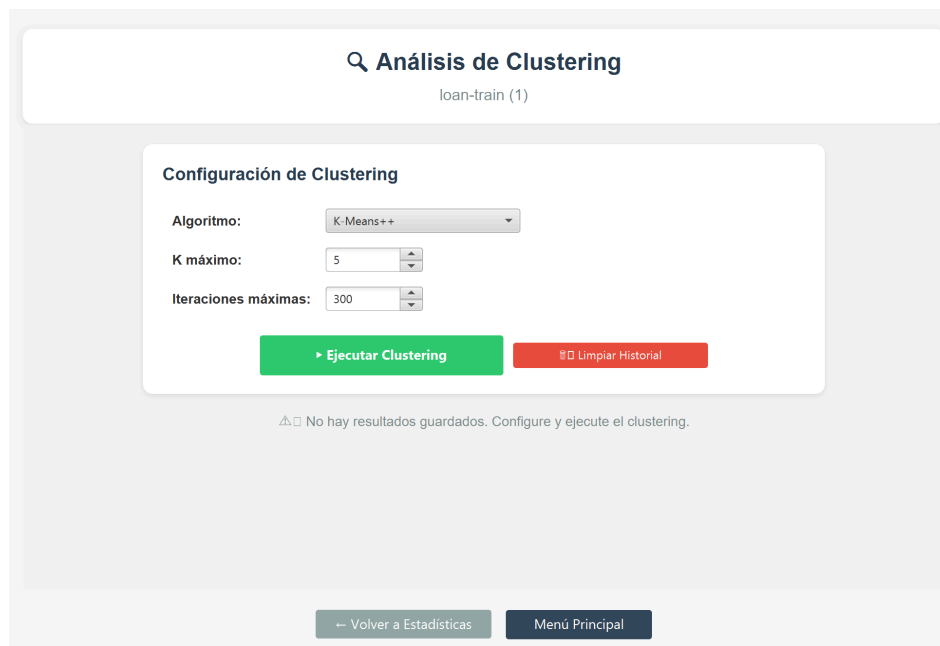
- idEncuesta: ID de la encuesta

#### 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



### 3.3.14. 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  $K \leq$  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 getUsersPorGrupo() 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

### 3.3.15. 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

### 3.3.16. ExportarCSVView

**Exportar Encuesta a CSV**  
Exporte los datos de una encuesta a formato CSV

**Formato de exportación:**

- Primera fila: nombres de columnas (ID\_Usuario + preguntas)
- Una fila por usuario que respondió la encuesta
- Compatible para re-importación

Seleccione la encuesta a exportar:

Seleccione una encuesta...

Guardar como:

Seleccione ubicación para guardar... **Examinar**

**Exportar a CSV** Cancelar

**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

## 4. Estructuras de datos y algoritmos utilizados

---

### 4.1 Estructuras de datos

#### 4.1.1 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  $\rightarrow$  *Encuesta.getPreguntas()*
- Las listas de respuestas de un usuario a una encuesta  $\rightarrow$  *Map<String, List<Respuesta>>* en *UsuarioAdmin* y *UsuarioRespondedor*
- Las filas leídas de un CSV en *LectorCSV*  $\rightarrow$  *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.

#### 4.1.2 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)*

#### 4.1.3 Set, HashSet y LinkedHashSet

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

- **HashSet**, por ejemplo:
  - En preguntas de tipo *Ordinal* para definir opciones válidas  $\rightarrow Set<String>$  *opciones*
  - En respuestas de categoría múltiple  $\rightarrow Set<String>$  *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.

#### 4.1.4 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  $\rightarrow String[]$
  - Cabeceras  $\rightarrow String[]$
- En *ResultadoClustering*
  - Grupos por usuario  $\rightarrow int[]$  *groups*
  - Centros de clúster  $\rightarrow Object[][]$  *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

#### 4.1.5 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.

## 4.2 Algoritmos

### 4.2.1 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.

#### 4.2.2 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.

### 4.2.3 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.

### 4.2.4 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  $\times$  preguntas, es decir, su coste es  **$O(U \cdot P)$**  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  **$O(U \cdot P)$** , 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.

#### 4.2.5 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.

#### 4.2.6 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.

#### 4.2.7 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.

## 5. Algoritmos de Clustering

---

### 5.1 Idea general

En este apartado se 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.

### 5.2 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**.
- Metadatos:
  - Nombre del algoritmo 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**.

## 5.3 AlgoritmoClustering

La capa del clustering sigue el patrón **estrategia** que nos permite encapsular diferentes algoritmos en una misma interfaz (*AlgoritmoClustering*). Esto facilita cambiar de algoritmo 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.

## 5.4 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.

### Algoritmo 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
  - Categoría simple: 0/1
  - Categoría múltiple: distancia tipo Jaccard
  - Texto libre: similitud basada en Levenshtein
- El usuario asigna el centro más cercano →  **$O(k \cdot m)$**  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 ( $b$ )
  - $\text{silhouette} = (b - a) / \max(a, b)$
- 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:  
 **$O(\text{maxIter} \cdot n \cdot k \cdot m)$**
- 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.

## 5.5 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 **distancia<sup>2</sup>**.

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  $O(k \cdot n)$  pero suele reducir el número de iteraciones necesarias para converger y mejora la calidad de la solución final.

## 5.6 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*

### Algoritmo 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:  
 **$O(\text{maxIter} \cdot k \cdot (n/k)^2 \cdot m)$**
- 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.

## 5.7 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, esto se explica en el punto 3.2.5.

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 ( $1 - \text{intersección/unión}$ ).
  - **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  $i$  y 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 código 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 método *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  $k = 2, 3, \dots, k_{max}$ .

### 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),
- **Inercia** (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

## 6. Jocs de proves

---

### 6.1 TestGestorEncuestas

#### 6.1.1 Objetivo de la prueba

Probar todas las operaciones *CRUD* (*Create, Read, Update, Delete*) del *GestorEncuestas*:

- Creación de encuestas con título y descripción
- Listado y consulta de encuestas existentes
- Modificación de datos de encuestas
- Eliminación de encuestas
- Gestión completa de preguntas (añadir, modificar, eliminar)
- Verificación de existencia de encuestas
- Estadísticas del sistema

#### Clases

- *GestorEncuestas* (clase principal del gestor)
- *Encuesta* (manipulamos indirectamente)
- Todas las subclases de *Pregunta*: *Numerica*, *Libre*, *CategoriaSimple*, *Ordinal*, *CategoriaMultiple*

#### 6.1.2 Otros elementos integrados

- **Scanner**: para interacción con el usuario
- **List, Set, Map**: estructuras de datos de Java utilizadas para gestionar colecciones.
- **Jerarquía de Pregunta**: prueba la creación y manipulación de diferentes tipos de preguntas

#### 6.1.3 Archivos de datos necesarios

No se requiere ningún archivo externo, toda la información se genera de forma interactiva durante la ejecución.

#### 6.1.4 Valores estudiados

##### Enfocamiento:

Caja Negra – Particiones equivalentes:

##### 1. Títulos de encuesta

- Títulos válidos no vacíos
- Títulos con espacios al inicio o al final (verificar `trim`)
- Títulos muy largos

## 2. Preguntas numéricas

- Sin límites (min = null, max = null)
- Solo mínimo definido Solo máximo definido
- Ambos límites definidos
- Límites inválidos (min > max)

## 3. Preguntas de texto

- Longitud por defecto (1000 caracteres)
- Longitud personalizada
- Longitud cero o negativa (debe provocar error)

## 4. Preguntas categóricas

- Con 2 opciones (caso mínimo)
- Con varias opciones (3–10)
- Opciones duplicadas
- Opciones vacías

## 5. Categoría múltiple

- $maxSelecciones \leq$  número de opciones
- $maxSelecciones >$  número de opciones (inválido)
- $maxSelecciones = 1$  (equivale a categoría simple)

Caja Blanca – Cobertura de caminos

- Modificación con valores *null*: comprobar que los valores anteriores se mantienen
- Eliminación con confirmación: probar confirmación y cancelación
- Listas vacías: comprobar comportamiento sin encuestas/preguntas
- Índices de pregunta: fuera de rango, válidos y en los límites (0, size-1)

### 6.1.5 Efectos estudiados

#### - Interfaz visual

- Uso de caracteres *Unicode* para cabeceras
- Líneas separadoras
- Iconos
- Identación y formato tabular

#### - Flujos de navegación

- Menú principal con 9 opciones
- Submenú de gestión de preguntas
- Opción de retorno “← Volver”
- Confirmaciones en operaciones destructivas (eliminación)

#### - Validación de entrada

- Manejo de enteros inválidos (método *leerEntero* con bucle)
- Gestión de líneas vacías (*trim*, *isEmpty*)
- Confirmaciones tipo **s/n**

- **Conservación de información**
  - IDs generados automáticamente
  - Estado consistente durante toda la sesión
  - Contadores actualizados (por ejemplo, número de preguntas)
- **Mensajes informativos**
  - Éxito
  - Error
  - Advertencias
  - Información general

### **6.1.6 Operativa**

#### **1. Crear una encuesta (Opción 1)**

- Título: “Satisfacción del servicio”
- Descripción: “Evaluación de la calidad”
- Añadir preguntas → Sí
- Pregunta numérica: “Puntuación del 0 al 10” (min=0, max=10)
- Pregunta de texto: “Comentarios” (longitud=500)
- Finalizar: Opción 6

#### **2. Listar encuestas (Opción 2)**

- Verificar que aparece la encuesta recién creada
- Comprobar el recuento total

#### **3. Ver detalle (Opción 3)**

- Introducir ID
- Revisar que se muestran todas las preguntas

#### **4. Gestionar preguntas (Opción 6)**

- Añadir categoría simple
- Modificar una pregunta existente
- Eliminar una pregunta
- Confirmar los cambios con “Ver preguntas”

#### **5. Modificar encuesta (Opción 4)**

- Cambiar solo el título
- Dejar descripción vacía
- Verificar el resultado

## 6. Estadísticas (Opción 8)

- Verificar conteos internos

## 7. Eliminar encuesta (Opción 5)

- Probar cancelación (no escribir “CONFIRMAR”)
- Probar eliminación correcta (escribir “CONFIRMAR”)

### Casos de error

- ID inexistente
- Números inválidos en cualquier menú
- Confirmaciones incorrectas
- Rangos numéricos inválidos (min > max)

## 6.2. TestGestorUsuarios

### 6.2.1 Objetivo de la prueba

Validar todas las operaciones del gestor de usuarios:

- Creación de usuarios respondedores (con y sin contraseña)
- Creación de administradores
- Consulta y listado
- Verificación de tipo y existencia
- Consulta de respuestas
- Estadísticas de actividad

### Clases cubiertas

- *GestorUsuarios*
- *Usuario* (abstracta)
- *UsuarioRespondedor*
- *UsuarioAdmin*
- *Respuesta* (consulta indirecta)

### 6.2.2 Elementos adicionales

- Herencia: navegación por la jerarquía
- Polimorfismo con *instanceof*
- Uso de *List* y *Set* para almacenar usuarios y respuestas



### 6.2.3 Archivos de datos necesarios

Ninguno, el sistema funciona completamente en memoria.

### 6.2.4 Valores estudiados

#### Enfocamiento:

Caja Negra – Particiones equivalentes

- **IDs de usuario**
  - IDs válidos y únicos (alfanuméricos)
  - IDs duplicados (deben rechazarse)
  - IDs vacíos o solo espacios
  - IDs con caracteres especiales
- **Nombres**
  - Nombres completos válidos
  - Nombres con múltiples espacios
  - Nombres vacíos
- **Passwords**
  - Contraseñas normales
  - Contraseñas vacías (caso import CSV)
  - Contraseñas extremadamente cortas o largas
- **Tipos de usuario**
  - Respondedor con contraseña
  - Respondedor sin contraseña
  - Administrador
- **Valores límite**
  - Listas vacías
  - Usuarios sin encuestas respondidas
  - Usuarios con múltiples respuestas

### 6.2.5 Efectos estudiados

- **Separación por tipo**
  - Listado independiente de Administradores y Respondedores
  - Iconos distintos
  - Información específica por tipo
- **Actividad**
  - Número de encuestas respondidas
  - Listado de respuestas
  - Ranking de usuarios más activos (Top 3)

- **Validaciones**
  - Detección de IDs duplicados
  - Verificación del tipo (isAdmin)
  - Existencia del usuario
- **Estadísticas**
  - Promedios (encuestas/usuario, respuestas/encuesta)
  - Totales agregados
  - Ordenación por actividad

### 6.2.6 Operativa

- **Crear usuarios**
  - Admin: id=admin1
  - Respondedor: user1
  - Respondedor: user2
  - Respondedor sin pass (csv001)
- **Listar usuarios**
  - Verificar separación por tipo
- **Buscar usuario**
  - Examinar user1
  - Examinar admin1
- **Verificar si es admin**
  - admin1 → Sí
  - user1 → No
- **Verificar existencia**
  - user1 → existe
  - user999 → no existe
- **Ver respuestas**
  - Usuario sin respuestas
  - (Requiere haber ejecutado TestGestorRespuestas antes)
- **Estadísticas**
  - Totales
  - Promedios
  - Top 3
- **Casos de error**
  - Crear usuario con ID duplicado
  - Buscar ID vacío
  - Pedir respuestas de un admin

## 6.3. TestGestorRespuestas

### 6.3.1 Objetivo de la prueba

Validar la integración completa del sistema:

- Respuesta a preguntas individuales
- Respuesta a encuestas enteras
- Consulta por usuario y por encuesta
- Validación según tipo de pregunta
- Gestión de índices

### Clases probadas

- *GestorRespuestas*
- *Respuesta*
- Integración con *GestorUsuarios* y *GestorEncuestas*

### 6.3.2 Integración

Este driver actúa como un test de integración completo.

Usa:

- *GestorEncuestas*
- *GestorUsuarios*
- Todas las clases de tipo *Pregunta*
- *Usuario* y derivados
- *Encuesta*

### 6.3.3 Archivos de datos necesarios

Ninguno, el método *crearDatosPrueba()* genera automáticamente:

- user1, user2, admin1
- Una encuesta “Encuesta de Satisfacción” con 4 preguntas:
  - Numérica
  - Categoría simple
  - Texto libre
  - Categoría múltiple

### 6.3.4 Valores estudiados

#### Validación por tipo de pregunta:

- **Numérica**
  - Valores dentro del rango
  - Valores fuera de rango
  - Enteros y decimales
  - Entrada no numérica
- **Texto libre**
  - Texto corto
  - Texto de límite (500 chars)
  - Texto excedido
  - Texto vacío
- **Categoría simple**
  - Opción válida
  - Opción fuera de rango
  - Respuesta nula
- **Categoría múltiple**
  - Selecciones válidas
  - Exceso de selecciones
  - Lista vacía
  - Duplicados
  - Índices inválidos
- **Obligatoriedad:**
  - Preguntas obligatorias sin responder
  - Opcionales sin responder
  - Combinación de ambas

#### Escenarios de integración:

- **Respuesta individual**
  - Usuario válido + encuesta válida + pregunta válida
  - Usuario inexistente
  - Encuesta inexistente
  - Índice fuera de rango
  - Pregunta que no pertenece a la encuesta
- **Respuesta completa**
  - Todas contestadas
  - Solo obligatorias
  - Falta una obligatoria
  - Un usuario respondiendo varias veces

- **Consultas**
  - Usuario con respuestas
  - Usuario sin respuestas
  - Encuesta con respuestas de varios usuarios
  - Encuesta sin respuestas

### 6.3.5 Efectos estudiados

- **Datos de prueba automáticos**
  - Mensajes informativos
  - IDs visibles
- **Interacción**
  - Prompts dependientes del tipo de pregunta
  - Indicadores de rango, límites y opciones
- **Visualización**
  - Formatos distintos por tipo
  - Separación por usuario
  - Agrupación por encuesta
- **Navegación compleja**
  - Usuario → encuesta → pregunta
  - Índices dinámicos
- **Validación en tiempo real**
  - Mensajes de error específicos
  - Correcciones en caso de error
  - Mensaje final de confirmación

### 6.3.6 Operativa

- **Flujo que vamos a seguir:**
  - Verificar datos generados
  - Responder pregunta individual
  - Responder encuesta completa
  - Ver respuestas de un usuario
  - Ver todas las respuestas
  - Obtener índice de pregunta
  - Ver estadísticas
- **Casos de error**
  - Responder como admin
  - Valor numérico fuera de rango
  - Exceso de opciones en selección múltiple
  - Omitir obligatorias
  - IDs inexistentes

## 6.4. TestGestorClustering

### 6.4.1 Objetivo de la prueba

Validar el funcionamiento completo del sistema de clustering y análisis de datos:

- Ejecución de algoritmos de clustering: *K-Means*, *K-Means++* y *K-Medoids*
- Comparación entre algoritmos con diferentes valores de K
- Cálculo de métricas de calidad: Silhouette e Inercia
- Conversión de respuestas a vectores numéricos
- Asignación de usuarios a grupos
- Medición de rendimiento temporal

### Clases probadas

- *GestorClustering* (clase principal)
- *Clustering* (patrón Strategy)
- *AlgoritmoClustering* (interfaz)
- *KMeans* (implementación concreta)
- *KMeansPlusPlus* (implementación concreta)
- *KMedoids* (implementación concreta)
- *ResultadoClustering* (contenedor de resultados)

### 6.4.2 Integración con otros módulos

Este driver actúa como un **test de integración completo del sistema**, ya que requiere:

- *GestorUsuarios*: creación de usuarios respondedores
- *GestorEncuestas*: creación de encuestas con diferentes tipos de preguntas
- *GestorRespuestas*: almacenamiento y recuperación de respuestas
- Jerarquía de *Pregunta*: Numérica, Ordinal, CategoríaSimple
- *UsuarioRespondedor*: usuarios que proporcionan los datos a agrupar
- *Encuesta*: estructura que contiene las preguntas a analizar

### 6.4.3 Archivos de datos necesarios

Ninguno, el método *crearDatosPrueba()* genera automáticamente:

- **1 encuesta** "Encuesta de Preferencias" con 4 preguntas:
  - Pregunta numérica: "Edad" (rango 18-80)
  - Pregunta numérica: "Ingresos" (rango 0-10000)
  - Pregunta ordinal: "Estudios" (Primaria, Secundaria, Universidad, Postgrado)
  - Pregunta categórica simple: "Vehículo" (Sí/No)

- **10 usuarios** con perfiles diversos:
  - user1 a user10
  - Edades entre 20 y 50 años
  - Ingresos entre 800 y 5000
  - Diferentes niveles educativos
  - Con y sin vehículo

#### 6.4.4 Valores estudiados

Caja Negra – Particiones equivalentes

- **Número de clusters (K)**
  - $K = 2$ : clustering binario (mínimo práctico)
  - $K = 3$ : clustering típico
  - $K = N/2$ : clusters medianos
  - $K \geq N$ : clusters vacíos o degenerados (caso límite)
  - $K = 1$ : todos en un grupo (caso trivial)
- **Número de iteraciones**
  - $\text{MaxIter} = 10$ : convergencia rápida
  - $\text{MaxIter} = 100$ : convergencia estándar
  - $\text{MaxIter} = 1000$ : convergencia garantizada
  - $\text{MaxIter} = 1$ : sin iteraciones (solo inicialización)
- **Tipos de datos en las preguntas**
  - Numéricas continuas: edad, ingresos (rangos amplios)
  - Ordinales: estudios (orden jerárquico)
  - Categóricas binarias: vehículo (Sí/No)
  - Combinaciones: todas juntas en la misma encuesta
- **Distribución de usuarios**
  - Grupos claramente separados: perfiles diferenciados
  - Grupos solapados: perfiles similares
  - *Outliers*: usuarios atípicos (ej: edad 50, ingresos 5000)
  - Distribución uniforme: usuarios homogéneos

Caja Blanca – Cobertura de caminos

- **Inicialización de centroides**
  - *K-Means*: selección aleatoria
  - *K-Means++*: selección probabilística (mejor distribución inicial)
  - *K-Medoids*: selección de puntos reales como centros
- **Proceso iterativo**
  - Asignación → Actualización: ciclo completo
  - Convergencia: centroides estables antes de  $\text{maxIter}$
  - Sin convergencia:  $\text{maxIter}$  alcanzado
  - Clusters vacíos: reasignación necesaria

- **Cálculo de métricas**
  - Silhouette > 0.7: clustering excelente
  - Silhouette 0.5-0.7: clustering aceptable
  - Silhouette < 0.5: clustering pobre
  - Inercia: suma de distancias cuadradas (debe disminuir con K)
- **Conversión de respuestas**
  - Valores numéricos: uso directo
  - Valores ordinales: mapeo a índices (0, 1, 2, 3)
  - Valores categóricos: codificación binaria (0/1)
  - Valores *null*: manejo de datos faltantes

#### 6.4.5 Efectos estudiados

- **Datos de prueba automáticos**
  - Generación de 10 usuarios con perfiles realistas
  - Diversidad en todas las dimensiones
  - Patrones detectables para validar clustering
  - Mensajes informativos de creación
- **Patrón Strategy**
  - Intercambiabilidad de algoritmos
  - Misma interfaz para diferentes implementaciones
  - Fácil extensión con nuevos algoritmos
- **Medición de rendimiento**
  - Tiempo de ejecución en milisegundos
  - Comparación de eficiencia entre algoritmos
  - Número de iteraciones hasta convergencia
- **Visualización de resultados**
  - Métricas de calidad claramente presentadas
  - Usuarios agrupados por cluster
  - Comparativa tabulada entre algoritmos
  - Formato estructurado y legible
- **Métricas de calidad**
  - Silhouette Score: cohesión intra-cluster vs separación inter-cluster
  - Inercia (WCSS): suma de distancias al cuadrado dentro de clusters
  - Número de iteraciones: indicador de convergencia
- **Comparación sistemática**
  - Evaluación con múltiples valores de K (2 a K\_max)
  - Comparación de los 3 algoritmos para cada K
  - Identificación del K óptimo (método del codo)



### 6.4.6 Operativa

#### Flujo que vamos a seguir:

1. Verificar datos generados

**Acción:** El sistema crea automáticamente los datos al iniciar

**Verificar:**

- Mensaje "10 usuarios creados"
- Mensaje "Encuesta lista para clustering"

2. Ejecutar K-Means (Opción 1)

**Parámetros:**

- K: 3
- MaxIter: 100

**Verificar:**

- Algoritmo: "KMeans"
- K: 3
- Iteraciones:  $\leq 100$  (debe converger antes)
- Silhouette: entre 0.3 y 0.8
- Inercia: valor positivo
- Tiempo:  $< 100ms$  para 10 usuarios
- 3 grupos con usuarios distribuidos

3. Ejecutar K-Means++ (Opción 2)

**Parámetros:**

- K: 3
- MaxIter: 100

**Verificar:**

- Silhouette generalmente  $>$  K-Means (mejor inicialización)
- Menos iteraciones hasta convergencia
- Distribución más equilibrada de usuarios

4. Ejecutar K-Medoids (Opción 3)

**Parámetros:**

- K: 3
- MaxIter: 100

**Verificar:**

- Centros son usuarios reales (no promedios)
- Mayor tiempo de ejecución que K-Means
- Robustez ante outliers

## 5. Comparar algoritmos (Opción 4)

**Parámetros:**

- K máximo: 5

**Verificar:**

- Ejecución para  $K = 2, 3, 4, 5$
- Comparativa de 3 algoritmos por cada K
- Silhouette disminuye con K alto (sobreajuste)
- Inercia siempre disminuye con K mayor
- Formato tabular de comparación

**Casos de error a probar**

- Valores de K inválidos
  - $K = 0$ : debe producir error o comportamiento indefinido
  - K negativo: debe producir error
  - $K > \text{número de usuarios}$ : clusters vacíos o errores
- *MaxIter* insuficiente
  - $\text{MaxIter} = 1$ : sin convergencia, resultados pobres
  - $\text{MaxIter} = 0$ : debe producir error
- Entrada de usuario
  - Valor no numérico en K o *MaxIter*: bucle de reintento
  - Valores muy grandes ( $K=1000$ ,  $\text{MaxIter}=10000$ ): tiempo excesivo
- Encuestas sin respuestas
  - Lista de usuarios vacía: debe manejar el caso gracefully
- Preguntas sin variabilidad
  - Todos los usuarios con mismas respuestas: clustering degenerado

**Escenarios avanzados**

- Convergencia
  - Verificar que el algoritmo se detiene cuando los centroides no cambian
  - Comprobar que iteraciones  $< \text{maxIter}$  indica convergencia exitosa

- Estabilidad
  - Ejecutar el mismo algoritmo varias veces
  - K-Means puede dar resultados diferentes (aleatorización)
  - K-Means++ debería ser más consistente
- Escalabilidad
  - Con 10 usuarios: respuesta inmediata
  - Tiempo de ejecución debe ser  $< 1$  segundo
- Calidad del clustering
  - Para los datos generados,  $K = 3$  debería dar mejor Silhouette
  - Grupos esperados: bajo/medio/alto ingresos + educación correlacionada
  - Vehículo y edad como factores secundarios

### Verificaciones de integración

- Con *GestorRespuestas*
  - Recuperación correcta de todas las respuestas
  - Manejo de preguntas obligatorias vs opcionales
  - Conversión correcta a vectores numéricos
- Con *GestorEncuestas*
  - Obtención correcta del orden de preguntas
  - Acceso a metadatos (rangos, opciones)
- Con *GestorUsuarios*
  - Filtrado de usuarios que respondieron
  - Exclusión de administradores
  - Manejo de listas vacías

### Resultados esperados

Para  $K=3$  con los datos de prueba:

- Silhouette Score: 0.4 - 0.7 (clustering razonable)
- Grupos típicos:
  - Grupo 0: jóvenes universitarios, ingresos bajos (user3, user7, user9)
  - Grupo 1: adultos con postgrado, ingresos medios (user1, user5)
  - Grupo 2: profesionales senior, ingresos altos (user2, user4, user6, user8, user10)

### Métricas de rendimiento:

- $K\text{-Means++} > K\text{-Means}$  en Silhouette
- $K\text{-Medoids}$  similar en calidad pero más lento
- Todos convergen en  $< 50$  iteraciones con datos limpios

## 6.5 Resumen de cobertura

- Cobertura de clases
  - *GestorEncuestas*
  - *GestorUsuarios*
  - *GestorRespuestas*
  - *GestorClustering*
  - *Encuesta*
  - *Usuario / UsuarioAdmin / UsuarioRespondedor*
  - Todas las clases de *Pregunta*
  - *Respuesta*
- Cobertura de casos de uso
  - Creación y gestión de encuestas
  - Creación y edición de preguntas
  - Gestión de usuarios
  - Respuestas individuales y completas
  - Consultas y estadísticas
  - Validaciones de entrada
  - Navegación y confirmaciones
- Método de prueba
  - Caja Negra: particiones equivalentes y valores límite
  - Caja Blanca: caminos críticos
  - Integración: enfoque *top-down* con datos generados

### Calidad de los drivers:

- Puntos fuertes
  - Interfaz clara y cuidada
  - Manejo robusto de errores
  - Cobertura amplia
  - Drivers fáciles de usar
  - Mensajes informativos y consistentes
- Mejoras posibles
  - Añadir *logs* automáticos
  - Ejecutar baterías de prueba automáticas
  - Comparar resultados esperados vs obtenidos