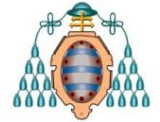




***Escuela de Ingeniería Informática***



UNIVERSIDAD DE OVIEDO

# SESIÓN 12 - PRÁCTICAS

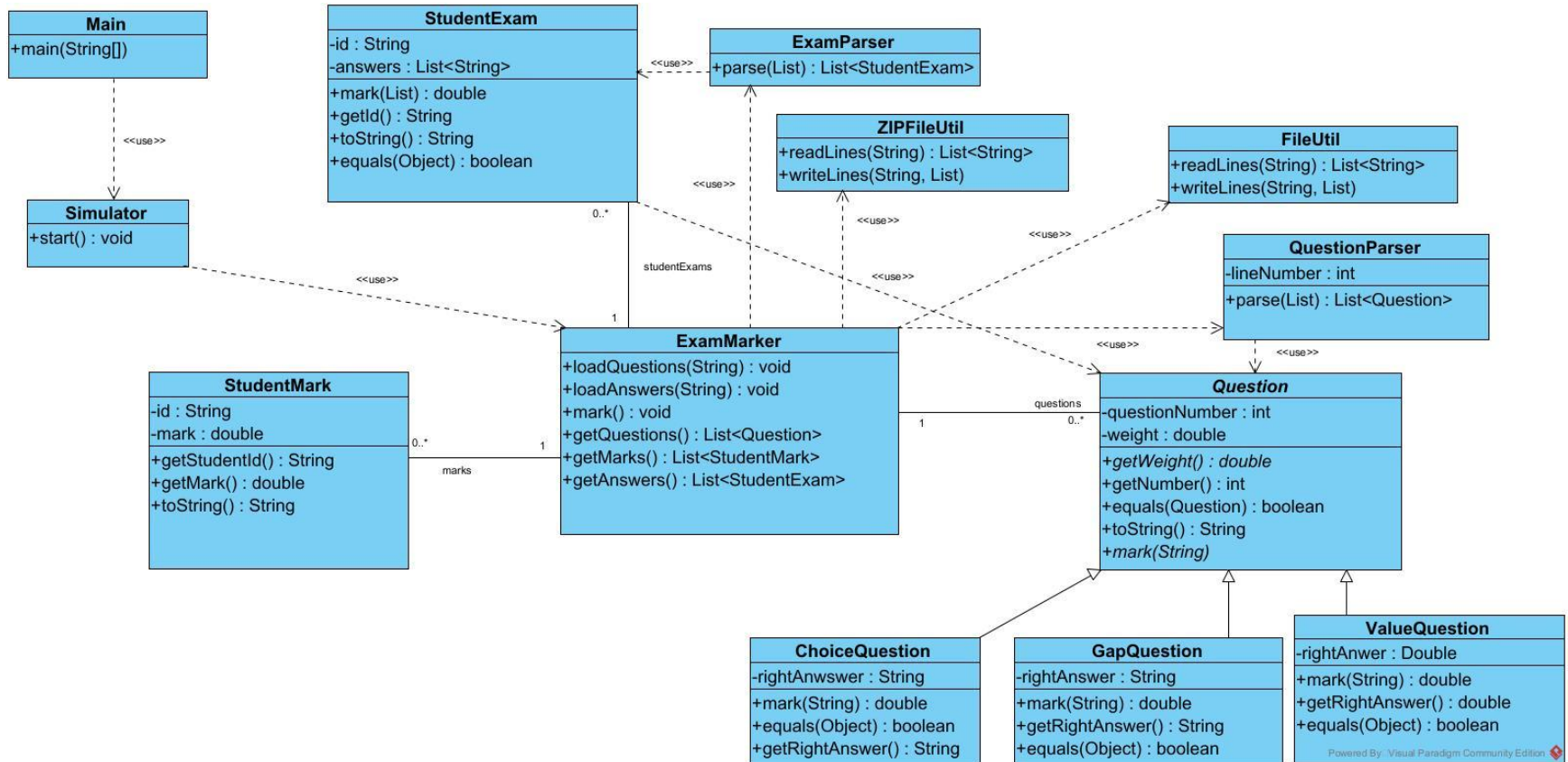
**Metodología de la programación**  
**Curso 2022-2023**

# Contenidos

---

- Proyecto ExamMarker.
  - UML
  - Carga fichero Questions
  - Carga fichero Answers
  - Gestión de excepciones
  - Operación Mark
  - Guardar a fichero
  - Test
  - Ordenación
  - UML completo

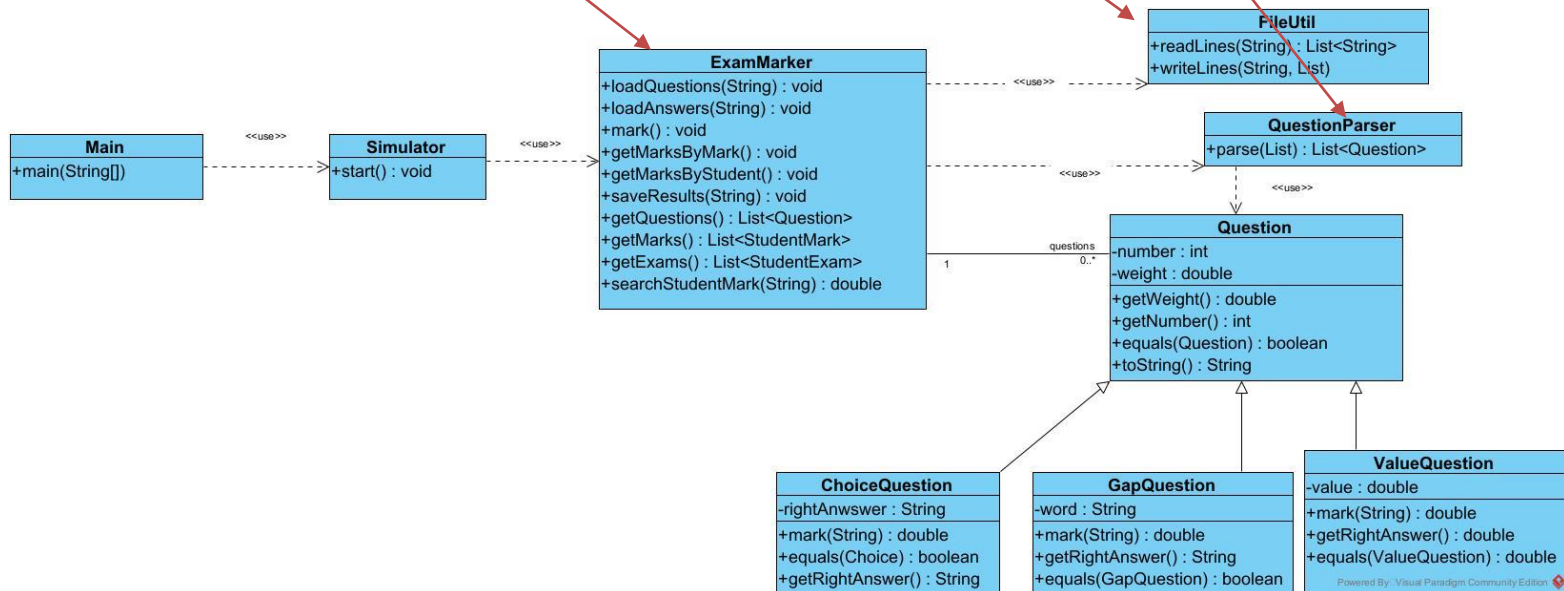
# ExamMarker. UML



# ExamMarker: carga preguntas

```
public void loadQuestions(String questionsFilename) throws ExamMarkerException {
    ArgumentsCheck.isTrue(questionsFilename != null );
    ArgumentsCheck.isTrue(!questionsFilename.isBlank() );
    → List<String> lines = readQuestionLines(questionsFilename);
    → List<Question> questions = new QuestionParser().parse( lines );
    → addQuestions( questions );
}

private void addQuestions(List<Question> quests) {
    for (Question q:quests) {
        questions.add(q);
    }
}
```



# LoadQuestions: lectura de fichero

```
public void loadQuestions(String questionsFilename) throws ExamMarkerException {  
    List<String> lines = readQuestionLines(questionsFilename);  
    List<Question> questions = new QuestionParser().parse( lines );  
    addQuestions( questions );  
}
```

```
private List<String> readQuestionLines(String questionsFilename) throws ExamMarkerException {  
    try {  
        return new FileUtil().readLines(questionsFilename);  
    } catch (FileNotFoundException e) {  
        throw new ExamMarkerException("Fichero de preguntas " + questionsFilename + "no encontrado");  
    }  
}
```

```
public class FileUtil {  
    public List<String> readLines(String inFileName) throws FileNotFoundException {  
        ArgumentsCheck.isTrue(inFileName != null && ! inFileName.isBlank() );  
        List<String> res = new LinkedList<>();  
        BufferedReader in = new BufferedReader(new FileReader(inFileName));  
        try {  
            try {  
                String line;  
                while ((line = in.readLine()) != null) {  
                    res.add(line);  
                }  
            } finally {  
                in.close();  
            }  
        } catch (IOException e) {  
            throw new RuntimeException("Error de lectura en " + inFileName);  
        }  
        return res;  
    }  
}
```

Fichero de texto

```
choice 1.0 b  
gap 0.5 stuff  
value 1.5 12.5
```

Recogemos  
FileNotFoundException  
para transformarla en  
ExamMarkerException con  
mensaje propio

res

Lista de Líneas

"choice

1.0

b"

"gap

0.5

stuff"

"value

1.5

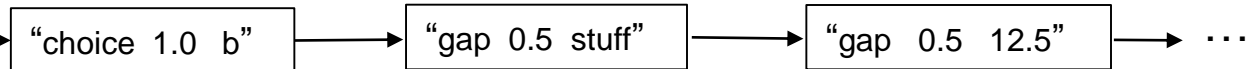
12.5"

...

# LoadQuestions: parser

```
public void loadQuestions(String questionsFilename) throws ExamMarkerException {  
    List<String> lines = readQuestionLines(questionsFilename);  
    List<Question> questions = new QuestionParser().parse( lines );  
    addQuestions( questions );  
}
```

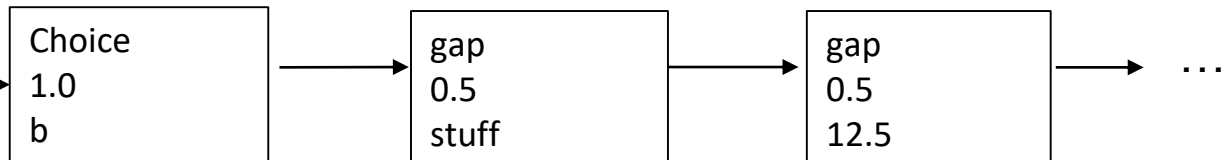
*lines*    Lista de Líneas



```
public List<Question> parse(List<String> lines) {  
    ArgumentsCheck.isTrue(lines != null);  
    List<Question> questions = new ArrayList<>();  
    for (String line : lines) {  
        lineNumber++;  
        try {  
            checkIsBlankLine(line);  
            Question q = parseLine(lineNumber, line);  
            questions.add( q );  
        } catch (InvalidLineFormatException e) {  
            Logger.log(e.getMessage());  
        }  
    }  
    return questions;  
}
```

Se recogen los  
errores del analizador  
Y se graban en el log

*questions*    Lista de questions



# LoadQuestions: parser

```
public class LineFormatException extends Exception {
    private static final long serialVersionUID = 1L;
    private int lineNumber;

    public LineFormatException(int lineNumber, String msg) {
        super(msg);
        ArgumentsCheck.isTrue(lineNumber > 0);
        this.lineNumber = lineNumber;
    }

    @Override
    public String getMessage() {
        return "INVALID LINE" + lineNumber + super.getMessage();
    }
}

private void checkIsBlankLine(String line) throws InvalidLineFormatException {
    if ( line.isBlank() )
        throw new InvalidLineFormatException(lineNumber, "Línea en blanco");
}

private Question parseLine(int ln, String line) throws InvalidLineFormatException {
    String parts[] = line.split("\t");
    checkThreeParts(parts);
    String type = parts[0];
    switch (type) {
        case "choice": return parseChoice(ln, parts);
        case "gap": return parseGap(ln, parts);
        case "value": return parseValue(ln, parts);
    }
    throw new InvalidLineFormatException(ln, "Tipo desconocido de pregunta " + type);
}

private double toDouble(String string) throws InvalidLineFormatException {
    try {
        return Double.parseDouble(string);
    } catch (NumberFormatException e) {
        throw new InvalidLineFormatException(lineNumber, "No es un valor numérico: " + string);
    }
}

private void checkThreeParts(String parts[]) throws InvalidLineFormatException {
    if ( parts.length != 3 )
        throw new InvalidLineFormatException(lineNumber, "La línea no tiene tres campos");
}
```

# ExamMarker: carga de respuestas

```
public void loadAnswers(String answersFilename) throws ExamMarkerException {
    ArgumentsCheck.isTrue(answersFilename!=null && !answersFilename.isBlank());
    List<String> lines = readAnswerLines(answersFilename);
    List<StudentExam> exams = new ExamParser().parse( lines );
    addExams( exams );
}
```

Caso en que se quisiera seguir cargando exámenes aunque haya error en una línea

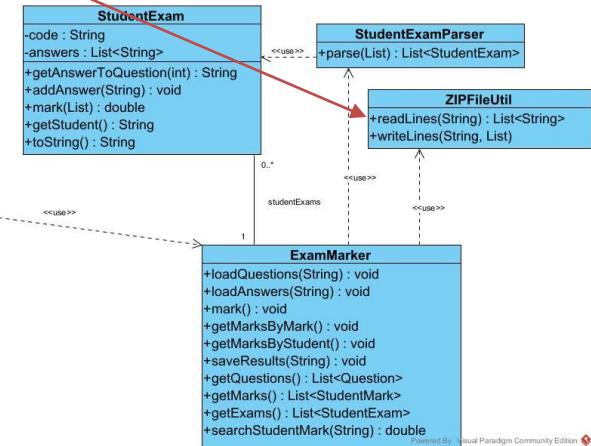
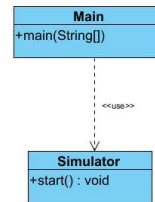
```
private List<String> readAnswerLines(String filename) throws ExamMarkerException {
    try {
        return new ZipFileUtil().readLines(filename);
    } catch (FileNotFoundException e) {
        throw new ExamMarkerException("Fichero " + filename + " no encontrado");
    }
}
```

Mensaje propio

```
private void addExams(List<StudentExam> exams) {
    for (StudentExam exam:exams) {
        try {
            addExam(exam);
        } catch (ExamMarkerException e) {
            Logger.Log(e.getMessage());
        }
    }
}
```

```
private void addExams(List<StudentExam> exams) throws ExamMarkerException {
    for (StudentExam exam:exams) {
        addExam(exam);
    }
}
```

```
private void addExam(StudentExam exam) throws ExamMarkerException {
    if (searchById (exam.getId()) == -1) {
        answers.add(exam);
    } else {
        throw new ExamMarkerException("Entrega repetida del alumno " + exam.getId());
    }
}
```



Si se lanza en cabecera implica que no sigue cargando más exámenes cuando hay error en una línea



# ExamMarker: gestión de excepciones

## Errores de Usuario

a) Errores al analizar las líneas del fichero (parsing errors). **Se invalida la línea y se continúa**

- Línea en blanco
- Número incorrecto de campos
- Tipos desconocidos
- Campos numéricos incorrectos
- ¿**Dónde se lanza ?** En algún método del Parser
- ¿**Qué excepción se lanza?** **InvalidLineFormatException**
- ¿**Qué mensaje lanza?** Error en línea X (y el mensaje concreto)
- ¿**Dónde se recoge?** En el método parse(), para poder continuar
- ¿**Cómo se maneja?**
  - Llamar al log con mensaje
  - INVALID LINE <line-number>: <message>

# ExamMarker: gestión de excepciones

## Errores de Usuario

b) **Añadir dos exámenes del mismo alumno** en el fichero de respuestas. Se indica que el programa debe terminar

- **¿Dónde se lanza?** Al añadir los exámenes en la aplicación, una vez leídos (`addExams()`)
- **¿Qué excepción se lanza?** `ExamMarkException`
- **¿Qué mensaje se lanza?** Entrega repetida del alumno `<identifier>`
- **¿Dónde se recoge?** En la interfaz, el programa debe finalizar (en start) (porque hay que terminar)
- **¿Cómo se maneja?** Manejador del usuario, Mensaje por consola

ERROR: `<mensaje>` Ejecute de nuevo

c) **Buscar nota de un alumno que no existe.** Se indica que el programa debe terminar

- **¿Dónde se lanza?** Operación específica, si no encuentra (`findMark()`)
- **¿Qué excepción se lanza?** `ExamMarkException`
- **¿Qué mensaje se lanza?**
- Nota no encontrada o estudiante desconocido `<iden`
- **¿Dónde se recoge?** En la interfaz, el programa debe finalizar (en start) (porque hay que terminar)
- **¿Cómo se maneja?** Manejador del usuario, Mensaje por consola

ERROR: `<mensaje>` Ejecute de nuevo

```
public StudentMark findMark(String id) throws ExamMarkerException {  
    for (StudentMark mark:marks) {  
        if (mark.getStudentId().equals(id)) {  
            return mark;  
        }  
    }  
    throw new ExamMarkerException("Nota no encontrada o estudiante desconocido");  
}
```

# ExamMarker: gestión de excepciones

## Errores de Usuario

d) **Fichero no encontrado.** Se indica que el programa debe terminar

- **¿Dónde se lanza?** Operación de lectura (readLines al crear el flujo)
- **¿Qué excepción se lanza?** **FileNotFoundException**
- **¿Qué mensaje se lanza?** Lo lanza el sistema. Hay que transformar en Fichero no encontrado

Transformamos FileNotFoundException en ExamMarkerException

```
private List<String> readQuestionLines(String questionsFilename) throws ExamMarkerException {  
    try {  
        return new FileUtil().readLines(questionsFilename);  
    } catch (FileNotFoundException e) {  
        throw new ExamMarkerException("Fichero de preguntas " + questionsFilename + "no encontra  
    }  
}
```

- **¿Dónde se recoge?** En la interfaz, el programa debe finalizar (en start) (porque hay que terminar)
- **¿Cómo se maneja?** Manejador del usuario, Mensaje por consola  
ERROR: <mensaje> Ejecute de nuevo

# ExamMarker: gestión de excepciones

## Errores de Programación y Sistema

d) Errores de operaciones de entrada /salida IOException (salvo FileNotFoundException)

Se consideran errores del sistema

- **¿Dónde se lanza?** En las operaciones de entrada/salida (operaciones del sistema)
- **¿Qué excepción se lanza?** **IOException que transformamos en RuntimeException**
- **¿Qué mensaje se lanza?** Error de entrada/salida
- **¿Dónde se recoge?** En la interfaz, el programa debe finalizar (en start) (porque hay que terminar)
- **¿Cómo se maneja?** Manejador del Sistema. Mensaje por consola para usuario, mensaje en log para administrador, pila de ejecución

```
private void handleSystemError(RuntimeException e) {  
    Console.WriteLine("ERROR IRRECUPERABLE: " + e.getMessage());  
    Logger.Log(e.getMessage());  
    Logger.Log(e);  
}
```

```
public List<String> readlines(String inFileName) throws FileNotFoundException  
{  
    ArgumentsCheck.isTrue(inFileName != null);  
    ArgumentsCheck.isTrue(!inFileName.isBlank());  
  
    List<String> res = new LinkedList<>();  
    BufferedReader in = new BufferedReader(new FileReader(inFileName));  
    try {  
        while (in.ready()) { // el fichero está listo para leer  
            res.add(in.readLine());  
        }  
        finally {  
            in.close();  
        }  
    } catch (IOException e) {  
        throw new RuntimeException("Error de lectura en " + inFileName);  
    }  
    return res;  
}
```

# ExamMarker: gestión de excepciones

## Errores de Programación y Sistema

e) Errores de programación. Producidos al validar

- Todos los parámetros deben ser distintos de null, incluidas las listas.
- Los Strings, además deben ser distintos de blanco.
- En el constructor de Question, no pueden ser negativos, ni el peso, ni el número de pregunta.
- En el constructor de StudentExam, el identificador debe ser de longitud 6
- **¿Dónde se lanza?** En las validaciones (ArgumentChecks y otras..)
- **¿Qué excepción se lanza?** **IllegalArgumentExpection**
- **¿Qué mensaje se lanza?** Depende del error
- **¿Dónde se recoge?** En la interfaz, el programa debe finalizar (en start) (porque hay que terminar)
- **¿Cómo se maneja?** Manejador del Sistema. Mensaje por consola para usuario, mensaje en log para administrador, pila de ejecución

# ExamMarker: gestión de excepciones

## Recogida de excepciones

1. En la capa de interfaz (start o menú si lo hay) **si se pide que el programa acabe**
  - Si hay menú se puede continuar con otra opción o acabar, según convenga
  - Si no hay menú se acaba después de manejar el error

Se propagan las excepciones hacia atrás hasta un método específico (start) cuya función principal sea gestionarlas.

```
public void start() {  
    simulateScenario();  
}
```

```
public void start() {  
    try {  
        simulateScenario();  
    } catch (RuntimeException e) {  
        handleSystemError(e);  
    } catch (ExamMarkerException e) {  
        handleUserException(e.getMessage());  
    }  
}
```

O bien

```
public void start() {  
    try {  
        simulateScenario();  
    } catch (RuntimeException e) {  
        handleSystemError(e);  
    } catch (Exception e) {  
        handleUserException(e.getMessage());  
    }  
}
```

Recoge el resto de las excepciones que lleguen aquí  
(FileNotFoundException, por ejemplo si no se ha transformado en ExamMarkerException más las ExamMarkerException)

# ExamMarker: gestión de excepciones

## Recogida de excepciones

### 2. En la capa de interfaz pero antes del final, si se pide mostrar error y continuar

```
private void simulateScenario() throws ExamMarkerException {
    ExamMarker ex = new ExamMarker();
    ex.loadQuestions( EXAM_MODEL_FILE );
    showQuestions(ex.getQuestions());
    ex.loadAnswers( ANSWERS_FILE );
    showExams(ex.getAnswers());

    ex.mark();

    showMarks( ex.getMarksByStudent(), " Students marks by ascending student id" );
    showMarks( ex.getMarksByDescendingMark(), " Students marks by descending mark" );
    showMarks( ex.getMarksByAscendingMark(), " Students marks by ascending mark" );

    ex.saveResults( RESULTS_FILE ); //OJO. graba la lista ordenada por orden ascendente de nota

    showMark(ex, "U00000"); // falla porque este UO no ha sido cargado, pero se recoge la excepción antes de llegar aquí
    showMark(ex, "U02021"); // SI se ejecuta
}

//Si cuando no encuentra un identificador, muestra mensaje pero continua (no acaba)
private void showMark(ExamMarker ex, String id) {
    try {
        double mark = ex.findMark(id).getMark();
        Console.println("Nota del estudiante " + id + ": " + mark);
    } catch (ExamMarkerException e){
        Console.println("Error. " + e.getMessage());
    }
}
```

# ExamMarker: gestión de excepciones

## Recogida de excepciones

3. En una clase de la capa de negocio, si se pide grabar en log y continuar

```
private void addExams(List<StudentExam> exams) {  
    for (StudentExam exam:exams) {  
        try {  
            addExam(exam);  
        } catch (ExamMarkerException e) {  
            Logger.Log(e.getMessage());  
        }  
    }  
}  
  
private void addExam(StudentExam exam) throws ExamMarkerException {  
    if (searchById (exam.getId()) == -1) {  
        answers.add(exam);  
    } else {  
        throw new ExamMarkerException("Entrega repetida del alumno " + exam.getId());  
    }  
}
```

Caso anterior pero  
ahora se pide que  
el programa  
acabe

```
private void addExams(List<StudentExam> exams) throws ExamMarkerException {  
    for (StudentExam exam:exams) {  
        addExam(exam);  
    }  
}  
  
private void addExam(StudentExam exam) throws ExamMarkerException {  
    if (searchById (exam.getId()) == -1) {  
        answers.add(exam);  
    } else {  
        throw new ExamMarkerException("Entrega repetida del alumno " + exam.getId());  
    }  
}
```



# ExamMarker: Mark

```
public void mark() {  
    marks.clear();  
    for (StudentExam exam: answers) {  
        double mark = exam.mark( questions );  
        marks.add( new StudentMark(exam.getId(), mark));  
    }  
}
```

```
public double mark(List<Question> questions) {  
    double value = 0.0;  
    for(int i = 0; i < questions.size(); i++) {  
        Question question = questions.get(i);  
        String answer = answers.get(i);  
        value += question.mark( answer );  
    }  
    return value;  
}
```

@Override

```
public double mark(String answer) {  
    ArgumentsCheck.isTrue(answer!=null && ! answer.isBlank());  
    return getRightAnswer().equals( answer )  
        ? getWeight()  
        : getWeight() * -0.2;  
    // Otra forma de hacer el return  
    if (getRightAnswer().equals( answer ) )  
        return getWeight();  
    else  
        getWeight() * -0.2;  
}
```

@Override

```
public double mark(String answer) {  
    ArgumentsCheck.isTrue(answer != null);  
    ArgumentsCheck.isTrue(!answer.isBlank());  
    return rightAnswer.equals( answer )  
        ? getWeight()  
        : 0.0;  
}
```

# ExamMarker: guardar a fichero

```
public void saveResults(String resultsFilename) {  
    MarkSerializer serializer = new MarkSerializer();  
    List<String> lines = serializer.serialize(marks);  
    new FileUtil().writeLines(resultsFilename, lines);  
}
```

ExamMarker

```
public List<String> serialize(List<StudentMark> marks) {  
    ArgumentsCheck.isTrue(marks != null);  
    List<String> lines = new ArrayList<String>();  
    for (StudentMark mark:marks) {  
        lines.add(mark.serialize());  
    }  
    return lines;  
}
```

MarkSerializer

```
public String serialize() {  
    return String.format("id %s --> nota %.2f", id, mark);  
}
```

StudentMark

%s para sustituir por un String  
%d para sustituir por un entero  
%f para sustituir por un doble o float  
%.2f (con 2 decimales)  
\n para que cambie de línea

# ExamMarker: Test parser

```
/*
 * Casos
 * 1- null en lugar de lista
 * 2- Lista vacía
 * 3- Lista con pregunta de cada tipo
 * 4- Lista con línea en blanco
 * 5- Lista con línea con tipo desconocido
 * 6- Lista con línea con formato de número incorrecto para peso
 * 7- Lista con línea con número de campos incorrectos para cada tipo choice
 * 8- Lista con línea con formato número incorrecto para respuesta correcta en tipo value
 */
```

```
/** 3-
 * GIVEN parser
 * WHEN se llama a parse lista con 3 líneas correctas con una pregunta de cada tipo
 * THEN devuelve lista con tres preguntas correspondientes
 */
```

```
@Test
```

```
public void ThreeQuestionsList() {
    result = parser.parse(list3);
    assertEquals(expectedList, result);
}
```

```
/** 6-
```

```
 * GIVEN parser
 * WHEN se llama a parse lista con 3 líneas correctas con una pregunta de cada tipo
 *      y una línea con formato de número incorrecto para el peso (de cada tipo)
 * THEN devuelve lista con tres preguntas correspondientes
 */
```

```
@Test
```

```
public void wrongNumberFormatInList() {

    list3.add("value 1.5g 12.5");
    result = parser.parse(list3);
    assertEquals(expectedList, result);
}
```

```
@BeforeEach
public void setUp() {
    parser = new QuestionParser();
    list = new ArrayList<String>();

    list3 = List.of("choice 1.0 a",
        "gap 0.5 stuff",
        "value 1.5 12.5"); CREA UNA LISTA INMUTABLE.
    // luego no se pueden añadir más
    list3 = new ArrayList<String>();
    list3.add("choice 1.0 a");
    list3.add("gap 0.5 stuff");
    list3.add("value 1.5 12.5");

    choiceQuestion = new ChoiceQuestion(1,1.0,"a");
    gapQuestion = new GapQuestion(2,0.5,"stuff");
    valueQuestion = new ValueQuestion(3,1.5,12.5);

    expectedList = List.of(choiceQuestion, gapQuestion, valueQuestion);
}
```

# ExamMarker: Test parser

```
test
├── uo.mp.s11.marker.parser.questionparser
│   └── ParseTests.java
├── uo.mp.s11.marker.service.exammaker
│   ├── LoadQuestionsTests.java
│   ├── LoadStudentExamsTests.java
│   └── MarkTests.java
```

```
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (!(obj instanceof Question))
        return false;
    Question other = (Question) obj;
    if (questionNumber != other.questionNumber)
        return false;
    if (weight == null) {
        if (other.weight != null)
            return false;
    } else if (!weight.equals(other.weight))
        return false;
    return true;
}
```

```
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (!super.equals(obj))
        return false;
    if (!(obj instanceof ChoiceQuestion))
        return false;
    ChoiceQuestion other = (ChoiceQuestion) obj;
    if (rightAnswer == null) {
        if (other.rightAnswer != null)
            return false;
    } else if (!rightAnswer.equals(other.rightAnswer))
        return false;
    return true;
}
```

```
@Test
public void wrongNumberFormatForValue() {
    inputList.add("value 1.5g 1h2.5");
    result = parser.parse(inputList);
    assertEquals(expectedList, result);
}
```

Para comparar objetos es imprescindible tener implementado el equals.


# ExamMarker: Test Serialize

```
/*
 * casos
 * 1- null en lugar de lista de notas
 * 2- Lista de notas vacía
 * 3- Lista de 3 notas
 *
 */
/**
 * GIVEN un serializador de notas
 * WHEN se invoca serialize con lista con 3 notas
 * THEN devuelve string con 3 notas
 *
 */
@Test

public void threeMarks() {
    markList.add(new StudentMark("U00001", 5.50));
    markList.add(new StudentMark("U00002", 8.50));
    markList.add(new StudentMark("U00003", 9.50));

    stringList.add("id U00001 --> nota 5,50");
    stringList.add("id U00002 --> nota 8,50");
    stringList.add("id U00003 --> nota 9,50");
    List<String> listResult = serializer.serialize(markList);
    assertEquals(stringList, listResult);
}
```

Mark debe tener implementado equals



# ExamMarker: Test loadQuestions

```
@BeforeEach
public void setUp() {
    examMarker = new ExamMarker();

    choiceQuestion = new ChoiceQuestion(1,1.0,"a" );
    gapQuestion = new GapQuestion(2, 0.5,"stuff");
    valueQuestion = new ValueQuestion(3, 1.5,12.5);

    expectedList = List.of(choiceQuestion, gapQuestion, valueQuestion);
}
```

Se puede usar List.of pero genera una lista inmutable (no se puede ampliar posteriormente)

Si se comparan objetos hay que redefinir el equals

```
/** 6
 * GIVEN examMarker
 * WHEN se llama a loadQuestions con fichero con 3 preguntas de los tres tipos
 * THEN deja el examMarker con lista con 3 preguntas
 */
@Test
public void with3QuestionsFile() {
    filename = "files/test/with3Questions.txt";
    try {
        examMarker.loadQuestions(filename);
        List<Question> resultList = examMarker.getQuestions();
        assertEquals(expectedList, resultList);
    } catch (FileNotFoundException e) {
        fail("Ha saltado FileNotFoundException");
    }
}
```

# ExamMarker: Test loadAnswers

```
/*
 * Casos
 * 1- null en lugar de fichero
 * 2- cadena vacía en lugar de fichero
 * 3- fichero no encontrado
 * 4- fichero vacío
 * 5- Fichero con una pregunta de cada
 * 6- Fichero con dos exámenes del mismo alumno
 */
```

StudentExam debe tener redefinido equals

```
/** CASO 5
 * GIVEN examMarker
 * WHEN se llama a loadAnswers con fichero con respuestas de 3 alumnos
 * THEN deja el examMarker con lista con 3 respuestas
 */
@Test
public void with3AnswersFile() {
    filename = "files/test/with3AnswersFile.gz";
    try {
        examMarker.loadAnswers(filename);
        List<StudentExam> resultList = examMarker.getAnswers();
        assertEquals(3, resultList.size());
        assertEquals(expectedList, resultList);
    } catch (Exception e) {
        fail("Ha saltado Exception");
    }
}
```

```
/** CASO 6
 * GIVEN examMarker
 * WHEN se llama a loadAnswers con fichero con respuestas de 4 alumnos, una de ellas repetida
 * THEN salta ExamMarkerException
 */
@Test
public void withRepeatedAnswersFile() {
    filename = "files/test/with3AnswersFile.gz";
    try {
        examMarker.loadAnswers("files/test/withRepeatedAnswersFile.gz");
        fail("Debería haber saltado excepción");
    } catch (ExamMarkerException e) {
        assertTrue(true);
    } catch (Exception e) {
        fail("Ha saltado excepción indebida");
    }
}
```

# ExamMarker: ordenación

Cómo se ordena:

1. Collections.sort(marks).

- Deja la lista ordenada por orden natural
- Necesita tener implementado compareTo de la Comparable<StudentMark>

```
public List<StudentMark> getMarksByStudent() {  
  
    Collections.sort(marks);  
  
    return (new ArrayList<StudentMark>(marks));  
  
    //Si nos piden lista ordenada pero sin modificar la original  
    List<StudentMark> sorted = new ArrayList<StudentMark>(marks);  
    Collections.sort(sorted);  
    return sorted;  
}
```

ExamMarker

```
@Override  
public int compareTo(StudentMark o) {  
    return (getStudentId().compareTo(o.getStudentId()));  
}
```

StudentMark



# ExamMarker: ordenación

Cómo se ordena:

2. Collections.sort(marks, new ByMarkAscendingComparator() ).

- Utiliza objeto comparador externo
- El comparador es una clase auxiliar que implementa Comparator<StudentMark>

```
public class ByMarkAscendingComparator implements Comparator<StudentMark> {  
  
    @Override  
    public int compare(StudentMark o1, StudentMark o2) {  
        int dif = ((Double) o1.getMark()).compareTo((Double) o2.getMark());  
        if (dif == 0)  
            return (o1.getStudentId()).compareTo(o2.getStudentId());  
        else  
            return dif;  
    }  
}
```

```
public class ByMarkDescendingComparator implements Comparator<StudentMark> {  
  
    @Override  
    public int compare(StudentMark o1, StudentMark o2) {  
        int dif = ((Double) o2.getMark()).compareTo((Double) o1.getMark()); // Por Nota descendente  
        if (dif == 0)  
            return (o1.getStudentId()).compareTo(o2.getStudentId()); //Ojo: por ID ascendente  
        else  
            return dif;  
    }  
}
```

# ExamMarker: UML completo

