

COURCE PROJECT

**Specialty**: SIT

**Faculty number**: 23221002

**Theme**: Creating a program in Java

Prepared by: Aleksander Bobylev Ivanovich

Checked by: Hristo Nenov

Varna

2025

Chapter 1.

* 1. The project is a simplified spreadsheet processor in Java. It allows the user to load tables from a prepared file, edit cell values, perform calculations using formulas, as well as parse data and save the table back to a file or to a new one.
  2. The goal of the project is to develop a console application for working with tabular data that supports:
* Opening reading and writing tables in text format
* Processing various types of data (integers, floating point numbers, strings, formulas)
* Editing specific cells by the user
* Calculating formulas with references to other cells
  1. Documentation structure
* Chapter 2. Overview of the subject area - covers basic concepts such as cell, formula, data types formulates the tasks and difficulties associated with implementing a spreadsheet
* Chapter 3. Design - describes the overall structure of the project, the logic behind the organization of classes and packages, and key diagrams and charts that explain the application architecture
* Chapter 4. Implementation and Testing - presents implementation details of the main classes, formula processing algorithms, testing scenarios and examples of interaction with the system
* Chapter 5. Conclusion - the results of the work done are summed up, the extent to which the set goals and objectives have been achieved and what further developments of the project

Chapter 2.

2.1 Basic definitions and concepts

* Cell - the main element of the table that stores the value entered by the user. It can contain an integer, a fractional number, a string, or a formula
* CellType - an enumeration (enum) that defines the content type: INTEGER, DOUBLE, STRING, FORMULA, EMPTY, ERROR
* Spreadsheet - a collection of cells organized as a two-dimensional "List<List<Cell>>”
  1. Definition of tasks and complexity of the problem to be solved

2.2.1 Parsing and storing data in a table structure

1. it is necessary to organize a two-dimensional structure (List<List<Cell>>) to store cell values ​​of different types: integers, fractions, strings and formulas
2. Processing empty cells, erroneous data and table boundaries requires additional checks and solutions for program stability

2.2.2 Support for different data types

1) Each cell (Cell) must correctly determine its type (CellType: INTEGER, DOUBLE, STRING, FORMULA, EMPTY, ERROR) based on the input text

2) Correct parsing

2.2.3 Calculating formulas with references to other cells

1) Formulas are processed as strings, starting with the = symbol

2) Also support for basic arithmetic operations

2.2.4 Implementation of the command line interface

1) Support for custom commands (edit, save, save as, print, open, exit,... )

2) Command processing

2.2.5 Working with files (reading/writing)

1) Implement saving table to file or new one

2.2.6 Difficulties and potential problems

1)Error handling

2) Extensibility

* 1. To achieve the set goals, the project uses classical methods of object-oriented programming (OOP), as well as basic design approaches that facilitate code maintainability and scalability.
* RawContent - a string representation of what the user entered before parsing
* Formula - an expression starting with the = symbol, containing references to other cells in like “R1C1”, “R2C2” format and arithmetic operations between them
* R1C1- Special syntax to specify a cell: “R<row number>C<column number>”
* DisplayValue - method for displaying a cell value in a readable form
* getValue() - method for getting already parsed cell value (Integer, Double, String)
* regEx - data type definitions
* matches - returns “true” if the entire string matches the regular expression
* parsingValue() - method for parsing value
* evaluateFormula() - calculating the formula
* getCellValueFromcase(String ref, spreadsheet forming) – parsing cell references
* loadFromLines(List<String>) - loading data
* saveToFile(String filename) – save to file
* print() - print table

Java constructs used

* Switch-case - for processing types, operators, commands\
* Try-catch - to catch errors
* Instanceof - check the type of a value (for example Number)
* Enum - description of fixed values ​​
* StringBuilder - efficient string building (saving CSV)
* List<List<Cell>> - table structure
* Scanner - console input
* Files.readAllLines and Files.write - working with files

SOLID principles

* S - Single Responsibility

1)Cell - logic for working with a single cell.

2)Spreadsheet - table logic.

3)Main - enter to programm

* O - Open/Closed

1)Only for classes Cell and spreadsheet good architecture( separated duties, table and cells can be expanded) but for another we need rewrite it is a point of improvement

* Liskov Substitution, Interface Segregation, Dependency Inversion

1)Not yet implemented, but possible in the future with increasing complexity of the architecture

Patterns Used

* Factory Method (simplest form) - determineType and parsingValue create the required values ​​from a string.
* Fail-Safe - Return 0 on errors in formulas or links - protection against crashes.
  1. User (functional) requirements and quality (non-functional) requirements

2.4.1 User (functional) requirements

* Opening files

The user must be able to load a table from a CSV file using the open <filename> command

* Editing cells

The user can change the value of any cell by coordinates using the edit <row> <col> <value> command.

* Saving a table

It is possible to save a table to a file using the save <filename> or saveas <newfilename> commands.

* Displaying a table

The user can display the current table on the screen as a formatted table using the print command

* Formula support

Cells can contain formulas, for example: “=R1C1 + R2C2”. Basic arithmetic operations +, -, \*, / are supported.

* Closing a table

The user can close a table by first confirming or rejecting this action with the close command

* Help

The help command is available, which displays a list of all commands and their descriptions

* Exiting the program:

The program is terminated by the exit command

2.4.2 Quality (non-functional) requirement

* Ease for use

All commands are simple, clear and entered from the console

* Scalability

The table is stored as a List<List<Cell>>, which allows dynamic processing of any number of rows and columns, limited only by memory

* Supportability (ease of maintenance)

The program implements OOP principles (encapsulation, division into classes)

* Reliability

Exception handling is implemented in many places (for example, when reading a file, when parsing formulas). In case of a formula error, the cell returns ERROR without disrupting the operation of the entire table.

* Security of out of bounds:

Prevented execution of incorrect commands. Table bounds checks protect against index overruns

Chapter 3.

3.1 General structure of the project, implemented packages

The project is implemented in Java and consists of several classes grouped into logical packages. The main goal of the architecture is to ensure modularity, ease of support and extensibility of the system.

* Spreadsheet package

1)Cell is the main class representing a table cell. It is responsible for defining the content type (integer, fractional, string, formula)

2)CellType -enumeration of possible cell content types. Used to correctly classify and process values

3) spreadsheet - a class that models a table as a two-dimensional structure (a list of rows, each of which contains cells). Contains methods for loading data, displaying the table, saving, accessing individual cells, and editing them

* Main class

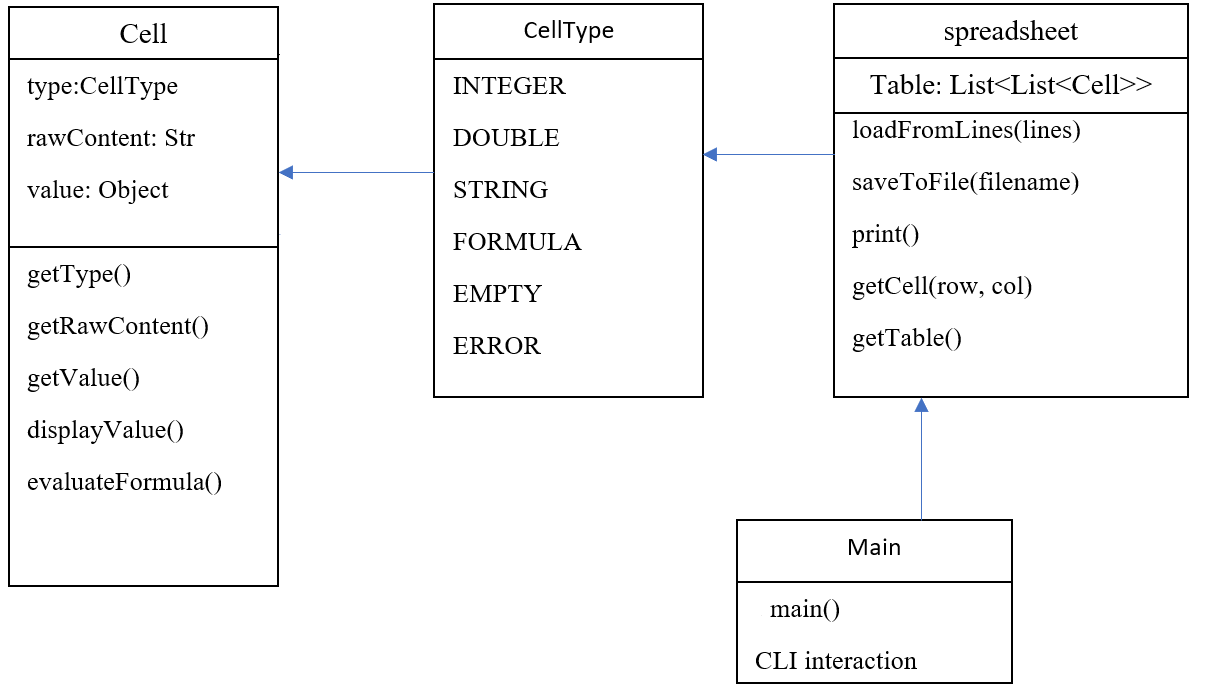
1)Implements the user interface from the console (CLI)

2)Supports commands open, edit, save, print, exit and others

3)Provides user interaction with the table

3.2 Diagrams and flow-charts

Flow-Chart 1



Most important code for design

private CellType determineType(String rawContent) {

if (rawContent.isEmpty()) return CellType.EMPTY;

if (rawContent.startsWith("=")) return CellType.FORMULA;

if (rawContent.matches("[-+]?\\d+")) return CellType.INTEGER;

if (rawContent.matches("[-+]?\\d+\\.\\d+")) return CellType.DOUBLE;

if (rawContent.matches("\".\*\"")) return CellType.STRING;

return CellType.ERROR;

}

private Object parsingValue() {

switch (type) {

case INTEGER:

return Integer.parseInt(rawContent);

case DOUBLE:

return Double.parseDouble(rawContent);

case STRING:

return rawContent.substring(1,rawContent.length() - 1);

case FORMULA:

return "FORMULA";

default:

return null;

}

1) Cell Class -Type Definition and Value Parsing

These methods determine the cell content type and parse the value, storing it in a convenient form for further use.

2)evaluateFormula() - metond for Evaluating Formulas

The logic for parsing and executing simple formulas is implemented here, including extracting values ​​from other cells (R1C1 format).

private Object evaluateFormula(String formula, spreadsheet forming) {

try {

formula = formula.substring(1); // убираем '='

String[] cases = formula.split(" ");

if (cases.length != 3) return "ERROR";

double val1 = getCellValueFromcase(cases[0], forming);

String operator = cases[1];

double val2 = getCellValueFromcase(cases[2], forming);

switch (operator) {

case "+": return val1 + val2;

case "-": return val1 - val2;

case "\*": return val1 \* val2;

case "/": return (val2 == 0) ? 0 : val1 / val2;

default: return "ERROR";

}

} catch (Exception e) {

return "ERROR";

}

}

case "\*": return val1 \* val2;

case "/": return (val2 == 0) ? 0 : val1 / val2;

default: return "ERROR";

}

} catch (Exception e) {

return "ERROR";

}

}

3) Spreadsheet Class - Method loadFromLines()

public void loadFromLines(List<String> lines) {

table.clear();

for (String line : lines) {

String[] parts = line.split(",");

List<Cell> row = new ArrayList<>();

for (String part : parts) {

row.add(new Cell(part));

}

table.add(row);

}

}

This method loads from file to table of rows, creates a two-dimensional List<List<Cell>> of cells, initializing the table object model.

4) Main Class - Handling the edit command

The edit command allows you to interactively update the contents of cells based on the row, row number, and column entered.

NewRow.set(col, new Cell(newValue));

System.out.println("Cell updated.");

} catch (NumberFormatException e) {

System.out.println("Invalid row/column number.");

}

break;

case "edit":

String[] Egitargs = argument.split("\\s+", 3);

if (Egitargs.length < 3) {

System.out.println("Usage: edit <row> <col> <value>");

break;

}

try {

int row = Integer.parseInt(Egitargs[0]) - 1;

int col = Integer.parseInt(Egitargs[1]) - 1;

String newValue = Egitargs[2];

if (row < 0 || row >= spreadsheet.getTable().size()) {

System.out.println("Row out of bounds.");

break;

}

List<Cell> NewRow = spreadsheet.getTable().get(row);

if (col < 0 || col >= NewRow.size()) {

System.out.println("Column out of bounds.");

break;

}

NewRow.set(col, new Cell(newValue));

System.out.println("Cell updated.");

} catch (NumberFormatException e) {

System.out.println("Invalid row/column number.");

}

break;

Chapter 4.

4.1 Implementation of classes

* Class Cell

The class is responsible for storing the contents of a cell, determining its type (int, string, formula,double) and calculating the value if it is a formula. Our more impornant things:

1. Method **determineType()**

The method analyzes the "raw" content of a cell (entered by the user) and determines what type of data it represents

private CellType determineType(String rawContent) {

if (rawContent.isEmpty()) return CellType.EMPTY;

if (rawContent.startsWith("=")) return CellType.FORMULA;

if (rawContent.matches("[-+]?\\d+")) return CellType.INTEGER;

if (rawContent.matches("[-+]?\\d+\\.\\d+")) return CellType.DOUBLE;

if (rawContent.matches("\".\*\"")) return CellType.STRING;

return CellType.ERROR;

}

1. Method **evaluateFormula()**

The most important method fo execute fornul. It receives values from other cells and performs operations on them (+, -, \*, /)

private Object evaluateFormula(String formula, spreadsheet forming) {

try {

formula = formula.substring(1); // deleting '='

String[] cases = formula.split(" ");

if (cases.length != 3) return "ERROR";

double val1 = getCellValueFromcase(cases[0], forming);

String operator = cases[1];

double val2 = getCellValueFromcase(cases[2], forming);

switch (operator) {

case "+": return val1 + val2;

case "-": return val1 - val2;

case "\*": return val1 \* val2;

case "/": return (val2 == 0) ? 0 : val1 / val2;

default: return "ERROR"; }

} catch (Exception e) {

return "ERROR";}}

1. Method **displayValue()**

This method is responsible for how the cell value will be displayed in the table. It takes into account the data type and, in the case of formulas, initiates their calculation in evaluateFormula()

public String displayValue(spreadsheet forming) {

if (type == CellType.EMPTY) return "";

if (type == CellType.ERROR) return "ERROR";

if (type == CellType.FORMULA) {

Object result = evaluateFormula(rawContent, forming);

return result != null ? result.toString() : "ERROR";}

return value != null ? value.toString() : "";}

* Class spreadsheet

The spreadsheet class is the core of the spreadsheet - it is responsible for storing, loading, editing and displaying all data. It implements the logic of working with cells through the List<List<Cell>> collection

1. Method **loadFromLines()**

This method initializes a table based on CSV rows. Each row is split into cells, each of which is turned into a Cell object

public void loadFromLines(List<String> lines) {

table.clear();

for (String line : lines) {

String[] parts = line.split(",");

List<Cell> row = new ArrayList<>();

for (String part : parts) {

row.add(new Cell(part));}

table.add(row);}}

1. Method **print()**

Doing a formats the output beautiful: 13 characters per cell, with separators(“ | ”)

public Cell getCell(int row, int col) {

if (row >= 0 && row < table.size()) {

List<Cell> targetRow = table.get(row);

if (col >= 0 && col < targetRow.size()) {

return targetRow.get(col);}

}

return new Cell(""); // empty cell }

* Class Main

The Main class implements the CLI (open, save, save as, edit, print, close, exit)

1. Case edit

I think it is most important case from all, it is show how it working

case "edit":

String[] Egitargs = argument.split("\\s+", 3); //split for 3 cases our input

if (Egitargs.length < 3) {

System.out.println("Usage: edit <row> <col> <value>");

break;

}

try {

int row = Integer.parseInt(Egitargs[0]) - 1; // index start from 0 bcs we subtract

int col = Integer.parseInt(Egitargs[1]) - 1;

String newValue = Egitargs[2]; // save new value

if (row < 0 || row >= spreadsheet.getTable().size()) { //the number rows of the table are not in the range of table

System.out.println("Row out of bounds.");

break;

}

List<Cell> NewRow = spreadsheet.getTable().get(row); // same for row like for cell

if (col < 0 || col >= NewRow.size()) {

System.out.println("Column out of bounds.");

break;

}

NewRow.set(col, new Cell(newValue)); //save to NewRow update cell with new value

System.out.println("Cell updated.");

} catch (NumberFormatException e) { //if error

System.out.println("Invalid row/column number. Please ensure the row and column are integers");

}

break;

The Main command has an edit command that allows you to change a cell by coordinates. If the input is correct, a new cell is created with a new value, replacing the old one. Basic checking for boundaries and data format is included.

4.2 Algorithms and optimizations

The project uses algorithms to process user input, classify data, navigate through a table, and calculate formulas.

* Processing formulas and navigating links

1. Removes the “=” character from the beginning of a string.
2. Split the formula three parts “R1C1 + R2C2”.
3. Takes values ​​from cells by reference (R2C2 → row 2, column 2).
4. Performs an operation between two values.

String[] cases = formula.split(" ");

double val1 = getCellValueFromcase(cases[0], forming);

double val2 = getCellValueFromcase(cases[2], forming);}

* Extracting values ​​by coordinates

1. The algorithm extracts values ​​from cells based on the R2C2 format.

if (!reference.matches("R\\d+C\\d+")) throw new Exception(...);

int row = Integer.parseInt(reference.substring(1, reference.indexOf('C'))) - 1;

int col = Integer.parseInt(reference.substring(reference.indexOf('C') + 1)) - 1;

* Multiple of types

1. The defineType() algorithm allows you to efficiently determine a type using a checking

* Security and bug-catching

1. Almost all potentially unstable sections of code (parseInt, getCellValueFromcase) are in try/catch

* Table output

1. The table display is implemented with alignment and prepared conversion of all values ​​to strings

System.out.print(String.format("%-13s", cellValue));

4.3 Planning, describing and creating test scenarios

* Loading and outputting a table

1. Scenario: Opening a CSV file with different data types
2. Expected result: Correct parsing of content, correct display of cells, alignment by width

* Editing cells
  1. Scenario: Command edit 1 2 5 change the cell of value.
  2. Expected result: Updated cell with new value.
* Calculating formulas

1. Scenario: Formula =R1C1 + R1C2, where both cells has numbers.
2. Expected result: Correct calculation of the sum and display of the result
3. Scenario (Error): Formula with invalid reference, for example =R100C5 + R1C1.
4. Expected result: Display "ERROR" without crashing the application.

* Saving and reopening

1. Scenario: Save table to file, reopen.
2. Expected result: Data is loaded in the same form as it was saved.

* Boundaries and Error Input

1. Scenario: Entering invalid commands or going out of table bounds (edit 99 99 value).
2. Expected result: User warning, no crashes.

Chapter 5

5.1 Conclusion

During the project implementation, a simple but functional spreadsheet processor was developed. What it does:

* Load tables from CSV files
* Edit individual cells and including formulas with references to other cells
* Handle different types of data: numbers, strings, empty values
* Save edited tables back to file or to a new one
* Provide basic protection from input errors

The program demonstrates the principles of modular design like a SOLID but in this project I have just S: The data processing logic is in the ***Cell*** class, the table structure is in the ***spreadsheet***, and user interaction is in ***Main***. (Single Responsibility Principle)

The main goals of my project were successfully achieved. The program has basic stability, is expandable due to the modular architecture and provides convenient interaction with the user through the command interface. This creates a foundation for further development.

5.2 Future improvements

I think that I have a decent program, but it is certainly not perfect. I have many ideas for improvement because there is no limit to perfection, but I will describe only the main problems as I see them;

* Cell dependencies

Ability to automatically update formula values ​​when dependent cells change.

* Managing table size

Ability to add and remove rows/columns, not just load a fixed table from a file.

* Support for complex formulas

Adding parentheses, more than one operator and standard math functions (e.g. SUM, AVG, MIN, MAX) like in Excel

* Сatching a context error

Indication of the error type (division by zero, invalid reference, syntax, etc.)

* Tabular output

In my case, I have a pre-made width in the table for the text, it is static, but I can make it dynamically expandable

* Tabular outputа for cell

I don't know if I have it right, but in theory when multiplying a number by a string it should come out 0, but for me it comes out 0.0

* Support for multi-level formulas with inter-cell dependencies

The current version implements basic work with formulas, but in the future it may be possible to implement the ability for one formula to refer to cells that themselves contain formulas. Like in example A1 = B1 + C1, where B1 = 2 + 3 and C1 = A2 \* 4.

* Saving and loading tables

Adding a function to save a table to a file and restore its state the next time the program is launched.

List of references

1. GeeksforGeeks — Expression Parsing in Java

<https://www.geeksforgeeks.org/expression-evaluation/>

1. **Simple math parser in Java (less than 200 lines of code**

**An example of implementing a simple parser of mathematical expressions in Java.**

[GitHub: Simple-Expression-Parser](https://github.com/javalc6/Simple-Expression-Parser)

1. This project is a simple implementation of a Java formula parser and calculator. It is inspired by EvalEx, but it is based

ondkellenb/formula-evaluator

1. Some same things

<https://github.com/jacklaaa89/ExpressionParser/tree/master>

1. Some same things

<https://github.com/aswath86/Spreadsheet-Calculator/tree/master>

1. Some same things for pars formula

https://developer.mescius.com/document-solutions/java-excel-api/docs/online/Features/ManageFormulas/formula-parser

1. <https://chatgpt.com/>
2. It's in russian but it really helped

https://metanit.com/java/tutorial/

Citation of literature

* SOLID - an acronym for the five principles of object-oriented programming
* OOP - Object-Oriented Programming
* CSV - Comma-Separated Values, Comma Separated Values a simple text format for storing tabular data. Each row is a row in the table, and values ​​are separated by commas
* CLI - Command-Line Interface, a method of interacting with a program through text commands entered into the console.