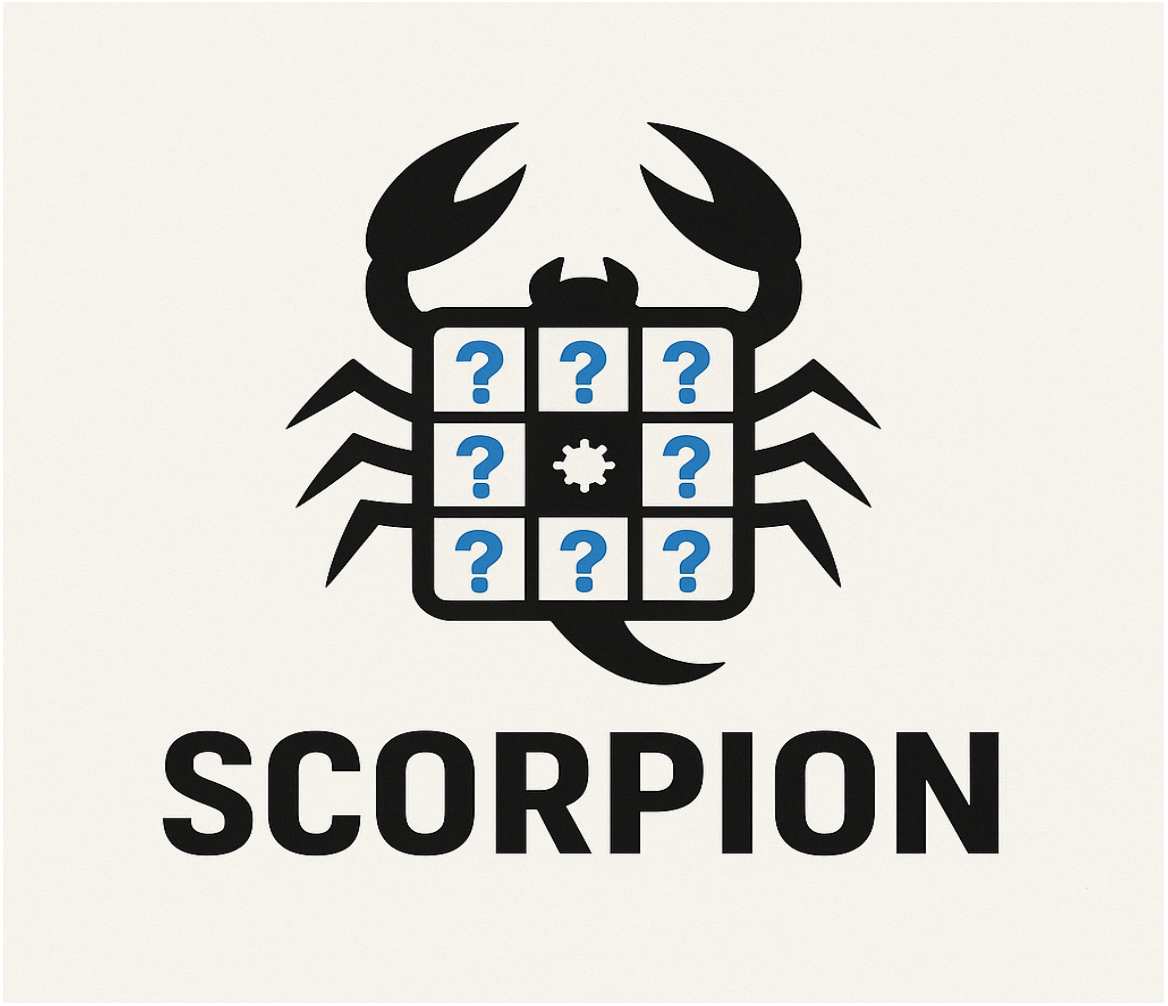
Scorpion Mine-sweeper

SRS



Participants:

Tamar Kohan, 330329038

Maria Abergel, 345210470

Rema Naamneh, 318695681

Hanen Naamneh, 213817927

Contents:

1.Introduction

The Scorpion Mine-Sweeper project is a logic-based game that merges the traditional Minesweeper concept with a question-and-answer component. This Software Requirements Specification (SRS) defines the system's complete set of functional and non-functional requirements

1.1 Purpose

The primary purpose of this SRS is to establish a shared understanding of the system's behavior among all stakeholders.

The document is intended for three main audiences:

- Developers and designers: To guide implementation, testing, and maintenance.
- Project managers and instructors: To verify the system meets academic and technical requirements.
- End users (players): To understand the game's intended functionality and user interface.

1.2 Scope

The software product, named Minesweeper Game System , is a two-player, turn-based game.

The system consists of the following integrated modules:

- Game Board Module: The core two-player, turn-based Minesweeper game.
- Question Management Module: Generates and manages multiple-choice questions.
- Data Management Module: Stores user data, history, and question banks using CSV files.
- Administrative Interface: Allows authorized users to manage (view, add, edit, delete) stored questions and game data.

The system's objectives include demonstrating a modular Java system following the MVC architecture and providing an engaging learning experience that integrates gameplay with knowledge testing.

Exclusions: The system will not include online multiplayer, advanced 3D graphics, or external database integration beyond CSV storage.

1.3 Definitions, acronyms and abbreviations

Term / Acronym	Definition
SRS	Software Requirements Specification
MVC	Model–View–Controller architecture pattern
CSV	Comma-Separated Values – format used for data storage
UI	User Interface
Mine	A hidden explosive cell on the game board
Tile / Cell	Individual square unit on the game board
Quiz Question	Multiple-choice question presented to the player during gameplay
Life Points	A numerical value representing a player’s remaining chances

GitHub	Online version control and collaboration platform
IntelliJ / Eclipse	Development environments used to implement the project

1.4 References

IEEE Std 830-1998 – IEEE Recommended Practice for Software Requirements Specifications.

Course guidelines for Software Engineering Project – Design and Quality in Information Systems, Semester Project Document (Hebrew original).

Java Development Kit (JDK 19) Documentation, Oracle, 2024.

GitHub Documentation – Using Git for Version Control, GitHub, 2025.

CSV File Handling API, OpenCSV Documentation, 2025.

1.5 Overview

The remainder of this document details the complete functional and non-functional requirements of the *Minesweeper Game System*. The SRS is organized to provide a clear, modular view of the system, ensuring traceability from user needs to implementation.

2. Overall description

2.1 Product perspective

The MineSweeper Scorpion system is a desktop application built according to the MVC (Model–View–Controller) software architecture.
The system is based on high modularity, allowing separation between the data layer, logic layer, and visual interface layer.
The system operates locally (Local Only) and preserves data in CSV files through a Data Management Module (SysData).

Concise MVC diagram:

- Model – includes the classes: GameBoard, Cell, Player, QuestionManager, SysData.
- View – the user interface (game board screen, question window, start menu, score and lives display).
- Controller – coordinates between player actions (clicks on cells, selecting difficulty level) and the logic in the Model, and updates the display in the View.

The product is part of a practical exercise in software architecture and system development in the field of Information Systems Engineering, and implements the rules of the classic MineSweeper game, with an educational extension through integrated trivia questions.

2.2 Main Functions

The system supports a complete set of main functions, implemented by the classes in the Model:

- Start a new game – initializes the board according to the selected difficulty level (Easy, Medium, Hard) and randomly places mines and special cells.
- Manage player actions – opens cells, places flags, presents questions, and provides immediate visual feedback.
- Score and life calculation – dynamic scoring according to the type of cell and action (positive/negative score, life loss or bonus).
- Trivia question management – displays a question when special cells are opened, verifies the answer, and updates score and lives accordingly.
- Statistical data storage – records player results, high scores, and general information in CSV files through the SysData module.
- End of game and ranking – presents a summary of points and lives, and declares victory or defeat.

2.3 User characteristics

- Players (End Users): Users aged 12 and above, possessing basic computer and mouse operation skills. The user must be able to read and understand simple trivia questions.
- System Developers: Information Systems Engineering students who apply OOP and MVC principles in Java.
- Testers and Instructors: Course instructors who use the system for demonstration and evaluation of the software development process and game execution.

2.4 Constraints

- Hardware limitations: The system requires a personal computer with at least 2 GB RAM and Windows 10 or newer operating system.
- Software limitations: The system is built and executed in Java (SDK 19) within the IntelliJ or Eclipse development environments.
- Data storage: There is no remote database – the system relies exclusively on local CSV files.
- Language and availability: The interface is available in Hebrew / English only; there is no multilingual support at this stage.
- Response time: Performance requirements are limited to a response time below 1 second per action and a maximum of 200 MB RAM usage (as defined in Section 3.3).

2.5 Assumptions and dependencies

- The user will run the system from a single personal computer (no simultaneous multi-user access).
- The required CSV files (Questions, Settings, High Scores) are located in the application folder and are accessible for reading and writing.
- There is no dependence on internet connection or any external server; all processing is entirely local.
- The system is operated by responsible adult users who will not damage the integrity of the data files.

- All files are saved in a standardized and up-to-date structure (SysData.csv, Questions.csv, Scores.csv).

3. Specific requirements

3.1 External interface requirements

3.1.1 User interfaces

The game provides a graphical user interface (GUI) built in Java. The main screen displays a grid-based game board, a score and life counter, and control buttons for starting, pausing, or resetting a game. When a player clicks a tile, it either reveals a value, a mine, or opens a question window with multiple-choice options.

The interface uses simple icons, color indicators, and dialog messages to provide feedback. Administrators can access an additional panel to manage questions—add, edit, or delete items. The layout is intuitive, responsive, and optimized for easy interaction using a mouse or keyboard.

3.1.2 Hardware interfaces

Mine-Sweeper runs as a stand-alone desktop application and requires only standard computer hardware. It operates on any system with Java JDK/JRE 19 installed. Input is provided via a keyboard and mouse. No special devices or external connections are required.

3.1.3 Software Interfaces

The system is based on internal interfaces between the following MVC modules:

Module	Interface	Interface Purpose
Controller ↔ Model	Direct Java calls (public methods)	Board update, cell verification, life and score management
Model ↔ SysData	Reading/writing in CSV files	Save player results and history

Model ↔ QuestionManager	Internal API for question operations	Display question and verify answer
View ↔ Controller	Event Listeners	Respond to user clicks and update the visual interface

All the interfaces are written in Java and implemented using objects and direct method calls, with no use of external libraries.

3.1.4 Communications interfaces

The system operates locally only (Offline).
There is no network communication with external servers or remote APIs.

The only communication mechanism includes:

- Reading and writing CSV files within the local file system (using Java I/O).
- Each file represents a separate data repository:
    - SysData.csv – system parameters and configuration.
    - Questions.csv – questions and answers.
    - Scores.csv – high scores and game results.
- File access is protected through exception handling (IOException), with default data loading in case of read failure.

3.2 Classes / Objects

3.2.1 Class / Object 1 – GameBoard

3.2.1.1 Attributes

3.2.1.1.1 boardId - Unique identifier for the board.
3.2.1.1.2 width / height – Number of rows and columns in the board.
3.2.1.1.3 cells : Cell[][] – Two-dimensional array of game cells.
3.2.1.1.4 difficulty : DifficultyLevel – Difficulty level {Easy, Medium, Hard}.
3.2.1.1.5 totalMines – Number of mines on the board.

3.2.1.1.6 totalQuestions – Number of question cells (if a QuestionManager exists).
3.2.1.1.7 initialLives / remainingLives – Number of lives at the beginning of the game and current amount.
3.2.1.1.8 remainingFlags – Number of flags remaining for the player.
3.2.1.1.9 currentScore – Player's accumulated score.
3.2.1.1.10 gameState : GameState – Game state {Ready, Running, Paused, Won, Lost}.
3.2.1.1.11 startTime / elapsedTime – Game start time and elapsed time in seconds.

3.2.1.2 Functions

3.2.1.2.1 initBoard(difficulty, width, height) – Creates a new board according to difficulty level and size.
3.2.1.2.2 placeMinesAndSpecialCells() – Places mines and special cells on the board.
3.2.1.2.3 revealCell(row, col) – Opens a cell and updates score / lives.
3.2.1.2.4 toggleFlag(row, col) – Marks or removes a flag from a cell.
3.2.1.2.5 handleQuestionResult(cell, isCorrect) – Updates score and lives according to the answer.
3.2.1.2.6 checkGameWon() – Checks victory condition.
3.2.1.2.7 checkGameOver() – Checks whether the game has ended.
3.2.1.2.8 getNeighbors(cell) – Returns the neighboring cells for mine-count calculation.
3.2.1.2.9 reset() – Resets the board to default values.

3.2.1.3 Messages

Receives from Player: startGame(), revealCell(), toggleFlag().
Sends to UI / Player: boardUpdated(), cellRevealed(), lifeChanged(), scoreChanged(), gameWon(), gameLost(), questionRequested().

3.2.2 Class / Object 2 – Cell

3.2.2.1 Attributes

3.2.2.1.1 row, col – The cell's position on the board.
3.2.2.1.2 contentType : CellType – {Empty, Mine, Number, Question, BonusLife, BonusPoints}.
3.2.2.1.3 adjacentMines – Number of mines surrounding the cell.
3.2.2.1.4 isRevealed / isFlagged – Open status and flag status.
3.2.2.1.5 basePoints – Base score for opening the cell.
3.2.2.1.6 questionId – Question identifier, if exists.

3.2.2.2 Functions

3.2.2.2.1 reveal() – Opens the cell and updates its status.
3.2.2.2.2 toggleFlag() – Changes the flag state.
3.2.2.2.3 setContent() – Initializes the cell type and its data.
3.2.2.2.4 isMine() / isQuestionCell() – Checks the content of the cell.

3.2.2.3 Messages

Receives from GameBoard: commands reveal() and toggleFlag().
Sends to GameBoard: mineExploded(), safeCellOpened(), questionCellOpened()

3.2.3 Class / Object 3 – QuestionManager

3.2.3.1 Attributes

3.2.3.1.1 questions – Question repository.
3.2.3.1.2 random – Random selection generator.
3.2.3.1.3 currentQuestion – Currently active question.
3.2.3.1.4 maxAnswerTime – Maximum time allowed for answering.

3.2.3.2 Functions

3.2.3.2.1 loadQuestions() – Loads questions from file / database.
3.2.3.2.2 getQuestion() – Retrieves a question by ID or randomly.

3.2.3.2.3 validateAnswer() – Checks the answer and returns the result.
3.2.3.2.4 getScoreDelta() – Calculates score change.

3.2.3.3 Messages

Receives from GameBoard: requestQuestion(), submitAnswer().
Sends to UI / Board: questionReady(), answerResult().

3.2.4 Class / Object 4 – Player

3.2.4.1 Attributes

3.2.4.1.1 playerId – Player identifier.
3.2.4.1.2 name – Player name.
3.2.4.1.3 currentScore – Current score.
3.2.4.1.4 remainingLives – Number of remaining lives.
3.2.4.1.5 difficultyPreference – Preferred difficulty level.
3.2.4.1.6 gamesPlayed / gamesWon / gamesLost – Player statistics.

3.2.4.2 Functions

3.2.4.2.1 startNewGame() – Initializes player data for a new game.
3.2.4.2.2 addScore(points) – Increases score.
3.2.4.2.3 loseLife() – Decreases lives.
3.2.4.2.4 isAlive() – Checks if any lives remain.
3.2.4.2.5 resetStats() – Resets player statistics.

3.2.4.3 Messages

Sends to GameBoard: startGame(), revealCell(), toggleFlag().
Receives from GameBoard / SysData: scoreChanged(), lifeChanged(), gameWon(), gameLost().
3.2.5 Class / Object 5 – SysData

3.2.5.1 Attributes

3.2.5.1.1 configPath – Path of the configuration file.
3.2.5.1.2 highScores – List of high scores.

3.2.5.1.3 statistics – Game statistics.

3.2.5.1.4 questionsSourcePath – Path of the questions file (if exists).

3.2.5.2 Functions

3.2.5.2.1 loadConfiguration() – Loads configuration.

3.2.5.2.2 saveConfiguration() – Saves configuration.

3.2.5.2.3 saveGameResult() – Saves game results.

3.2.5.2.4 loadHighScores() – Loads high scores.

3.2.5.2.5 loadQuestions() – Transfers the question repository to QuestionManager.

3.2.5.3 Messages

Receives from GameBoard / Player: save and load requests.

Sends to QuestionManager: questions after loading.

Sends to UI: list of high scores and results.

3.3 Performance Requirements

3.3.1 PR-1 – Game Initialization Time

Initialization of a new board (up to 16×16, 40 mines) ≤ 1 second in 95 % of cases; no more than 2 seconds in the worst case.

3.3.2 PR-2 – User Action Response Time

Opening a cell or placing a flag ≤ 0.5 second; question window ≤ 1 second.

3.3.3 PR-3 – Answer Processing Time

Checking an answer and returning the result ≤ 0.5 second.

3.3.4 PR-4 – Memory Usage

Maximum RAM usage < 200 MB during runtime.

3.3.5 PR-5 – Data Saving Latency

Saving game results ≤ 0.5 second.

3.3.6 PR-6 – Availability

The game supports continuous use of ≥ 3 hours without significant slowdown (≤ 20 % performance decrease).

3.4 Logical database requirements

The system does not use a traditional relational database,but it applies equivalent logical design principles through CSV-based storage handled by the SysData component. All information is structured, persistent, and verified according to consistent logical rules.

a) Types of Information Used by Various Functions

The system manages three main data structures:

1. Questions Data – Each question includes:

    Question ID (unique numeric identifier), question text (string), four possible answers, correct answer indicator, difficulty level (Easy / Medium / Hard), category or type of question

2. This dataset is used during gameplay to display random questions and verify player answers.

3. Game Results Data – Stores every completed session’s summary:

    Player name or identifier, date and time of the session, difficulty level, total score and remaining live, number of questions answered correctly, duration of the game

4. This dataset supports the *Game History* screen and performance evaluation.
5. Player Statistics Data – Tracks long-term achievements:

    Total games played, Highest score achieved, Average accuracy rate, Preferred difficulty level

6. These records help personalize feedback and display the top-scoring players.

All data are stored in structured CSV files managed by the SysData class.

b) Frequency of Use

- Questions Data are loaded once at application startup and used continuously throughout gameplay.
- Game Results Data are updated each time a session ends.
- Player Statistics Data are read and updated after every completed game.
- Administrative operations (add, edit, delete question) occur occasionally, only when instructors or authorized users update the question bank.

c) Accessing Capabilities

- All access to stored data is performed exclusively through the SysData API.
- The game logic and UI modules can read data but cannot directly modify CSV files.
- Write permissions for question management belong only to the administrator.
- Data are loaded synchronously to avoid inconsistent states, and validation is performed on every write operation to ensure correct format and data types.
- The system automatically creates missing CSV files at startup to guarantee reliability.

d) Data Entities and Their Relationships

Entity	Attributes	Relationships
Player	player ID, name, total games, best score	One Player → many Game Sessions
Game Session	session ID, date, difficulty, score, time elapsed	Each session → many Questions
Question	question ID, text, answers A–D, correct answer, difficulty	Questions are shared across multiple sessions
SysData	manager of read/write operations for all CSV files	Central controller that links all entities logically

A Player can participate in many Game Sessions.
Each Game Session uses multiple Questions randomly selected from the Question Bank.
Questions can be reused across different sessions and players.

e) Integrity Constraints

- Uniqueness: Each Question ID and each Game Session record must be unique.
- Non-Null Fields: Question text, options, and correct answer cannot be empty.
- Domain Limits: Difficulty must be one of {Easy, Medium, Hard}.
- Value Validation: Scores and lives must be positive integers; dates must be valid ISO format.
- File Integrity: Before each read or write, the system verifies file existence and consistency.
- Transaction Safety: If an error occurs during write, the operation is rolled back to prevent corruption.
- Data Consistency: All related records (player, game, question) must preserve referential consistency inside their CSV files.

f) Data Retention Requirements

All data persist on disk until explicitly deleted by the administrator.
There is no automatic expiration mechanism; this ensures complete history for analysis and grading purposes.
Backups can be made manually by copying CSV files from the project directory.
The system may later be extended to migrate these files into a relational database (e.g., MySQL) without changing the logical structure.

3.5 Design constraints

- The game runs locally on a personal computer without internet connection. The system is developed in Java using the MVC architecture.
- All data is stored in local CSV files only.
- No external libraries, databases, or servers are used.
- The design and documentation follow the IEEE 830 standard and university guidelines.
- The interface is simple, user-friendly, and suitable for all ages.

3.6 Software system attributes

3.6.1 Reliability

The Mine-Sweeper system must behave consistently in every game session.
Game data (scores, lives, questions, board state) are kept accurate and synchronized with CSV storage.
File operations use validation and error handling so missing or corrupted files do not crash the system.
Input validation prevents invalid data from being stored or processed.
Critical operations (cell reveal, scoring, questions) are tested under normal and boundary conditions, with a target of at least 99% error-free test runs.

3.6.2 Availability

The system is a standalone desktop Java application with no dependency on internet or external servers.
It runs on any computer with the required Java JDK installed.
All data (questions, history, results) are stored locally in CSV files and remain available after restart.
Unexpected termination does not damage stored data; availability depends only on the local machine’s readiness.

3.6.3 Security

Offline operation protects data from external access.
Only authorized users (administrators) can modify the question bank and configuration via the UI.
CSV records are validated before saving to preserve integrity.
Separation between UI, logic, and data modules reduces accidental overwriting or misuse of shared data.

3.6.4 Maintainability

Maintainability is ensured through a clear MVC structure.
Classes such as `GameBoard`, `Cell`, `Question`, and `SysData` have focused responsibilities and are documented.

Version control with GitHub supports collaboration and rollback.
Consistent coding style and modular design simplify debugging and future extensions (e.g., new levels or question types).

3.6.5 Portability

The system is implemented in Java and runs on Windows, macOS, and Linux without code changes.
It uses relative file paths and avoids OS-specific or native libraries.
It is distributed as a runnable JAR executable on any compatible Java Runtime.
The core logic can be reused on other platforms with minimal adaptation.