Rust Spreadsheet: Design and Software Architecture

Sourabh Verma, Sankalp Omkar, Eeshan Yadav
GitHub Repository

April 25, 2025

1 Design and Software Architecture

1.1 Additional Extensions Implemented

On top of the proposed extensions, we also implemented many extra extensions which are as follows:

1) 3 modes: Cell Selection Mode, Edit Mode, Command Mode (like Vim)

2) 3 ways to edit a cell:
   a) Select a cell and use formula bar
   b) Double click a cell to edit it (edit mode)
   c) Use Command Bar (Commands same as Terminal)

3) 6 methods to scroll:
   a) Scroll bars (mentioned in extensions)
   b) scroll wheel of mouse
   c) scroll button
   d) using keyboard arrows (can change cell selection and scroll along with it)
   e) `scroll_to` [cell] in command bar
   f) `w,a,s,d<amount>`

4) Inspired from Vim, we made sure that all features of our spreadsheet are accessible from only keyboard too i.e. it is completely keyboard operational.

5) Aside from supporting only CSV format (which stores the values of the cells) we also support FCSV format (which stores the formulae of the cells) and is compatible with excel.

6) Cut was added along with Copy, Paste and their shortcuts were also added.
Ctrl+E for Copy
Ctrl+R for Paste

Ctrl+T for Cut
The reason for non-standard key bindings is that Ctrl+X, Ctrl+C and Ctrl+V had pre-assigned meaning in `egui` (the GUI which we used).

7) Colour Picker was added which can change the colour of the spreadsheet. Many fun themes were also added such as "rainbow1, rainbow2, matrix1, matrix2, matrix3, love" with animations in all of these themes and "tr" for theme reset!

8) New commands like `goto` (changes the cell selection) and `w,a,s,d<amount>` were added which are extension of existing `w,a,s,d` by scrolling the spreadsheet with `<amount>` cells in respective direction.

1.2 Primary Data Structures

- **Cell:** Stores the value of a cell (which can be an integer, string, or error), the type of the cell (e.g., formula type with references), and a `HashSet` of all dependent cells.
- **CellName:** Cell references are not stored as strings (which take 24 bytes) but as compact 8-byte representations.
- **Spreadsheet:** A `HashMap<u32, Cell>` with an initial capacity of 1024 entries. The map grows dynamically as more cells are used.
- **FormulaType:** An `enum` representing different types of formulas, such as constants, ranges, and others.
- **Is_range:** A `Vec<bool>` that indicates whether each cell is within the range of any other cell.
- **Ranged:** A data structure that maps a cell to the start and end of any range it depends on.
- **SpreadsheetStyle:** Contains visual properties for the GUI, including cell colors, headers, columns, bars, and font sizes.
- **SpreadsheetApp:** Acts like a central application state, storing elements such as event listeners, undo/redo stacks, the currently selected cell or range, and various helper structures required by the GUI.

1.3 Module Interfaces

→ `main.rs` is the entry point of our program which has 2 features:
autograder
gui

→ The autograder part of extension contains 3 modules:
`parser.rs`

`utils.rs`
`scrolling.rs`

→ The GUI-related files are kept in a separate folder `src/gui/` and have 5 modules:
- `gui_defs.rs`
- `impl_helpers.rs`
- `render_gui.rs`
- `scroll_gui.rs`
- `utils_gui.rs`

→ The tests are kept in `src/test/test.rs` module.

1.4 Encapsulation Strategies

- The enums and struct definitions are kept in `gui_defs.rs`. This module acts as the central authority for data structure definitions within the GUI.
- Struct fields are accessible only within the GUI crate using `pub(in crate::gui)`, which prevents external manipulation and enforces encapsulation.
- Other modules within the GUI crate are kept private to hide implementation details and reduce coupling.
- This separation of concerns allows for maintainable, self-contained design where internal changes don't affect public interfaces.

1.5 Design Justification

The architecture of our Rust-based spreadsheet system is deliberately structured to balance performance, modularity, and ease of development. The following design choices contribute to a robust, maintainable, and extensible system:

- **Efficient Memory Usage:** The core spreadsheet logic uses a `HashMap` to store only active cells, significantly reducing memory usage for sparse spreadsheets. This optimization brings down memory consumption from approximately 1.5 GB to 40–50 MB in common use cases.
- **Performance-Optimized Range Handling:** Instead of scanning entire ranges (e.g., `SUM(B1:ZZZ999)`), we use a `ranged` map to efficiently track which cells are affected by ranges. This drastically reduces computation time in heavy range-based formulas from multiple seconds to a few milliseconds.
- **GUI Integration with `egui` and `eframe`:** The entire graphical interface is implemented using `egui` and `eframe`, which are lightweight, fast, and idiomatic Rust GUI libraries. This allows tight integration between frontend and backend logic, faster prototyping, and easy cross-platform builds.

- **Stress Testing Validation:** We performed extensive stress testing on large spreadsheets with complex formulas and dependencies. The system maintained low memory usage and consistently fast response times, validating the effectiveness of our design under real-world workloads.
- **Separation of Concerns and Modularity:** The system separates computational logic (cells, formulas, ranges) from presentation (styles, GUI state) and interaction logic (event handling, undo/redo, selection). This makes the codebase easier to understand, test, and extend.
- **Undo/Redo Functionality:** By maintaining application-level stacks for undo and redo operations, the user experience remains intuitive and forgiving, which is critical in interactive editing environments.
- **Rust Safety Guarantees:** Leveraging Rust's strong typing, memory safety, and ownership model ensures that the system avoids common bugs like use-after-free, null pointers, or data races, especially important in a multi-component architecture.
- **Scalable and Future-Proof:** The use of enums (e.g., `FormulaType`) and modular components makes it easy to extend the application with new features, such as custom functions, more visual styles, or collaborative editing.
- **Minimal Overhead by Default:** Because cells default to zero and are only instantiated when needed, the baseline footprint is extremely low, making this system suitable for embedded or constrained environments too.

Together, these design decisions provide a solid foundation for a spreadsheet engine that is efficient, user-friendly, and ready for future enhancements.

1.6 Design Modifications

The earlier design was inefficient in terms of both memory usage and computation time. A key bottleneck was the use of a full-size visited array for all cells, which led to significant overhead.

In the revised design, a `HashMap` with an initial capacity of 1024 is used to store only active cells. A cell is inserted into the map only when it is explicitly used (e.g., writing `A1 = 1` inserts `A1` into the map). Unused cells are implicitly treated as zero and are not stored. This optimization significantly reduced memory consumption from approximately 1.5 GB to around 40–50 MB in most practical scenarios. Additionally, this reduction in data size leads to faster iteration times.

For range calculations (e.g., `A2 = SUM(B1:ZZZ999)`), instead of iterating over the entire cell range from start to end, the system now iterates only over the keys in the `ranged HashMap`. It checks whether each cell falls within the

specified range and includes it in the computation if so. Since most cells are unused (implicitly zero), this approach avoids unnecessary work and dramatically improves performance—from 4–5 seconds down to approximately 0.3 seconds on moderately large but computation-heavy test cases.

1.7 Challenges Faced and Limitations

1.7.1 Proposed Extensions Not Implemented

All but one of the proposed extensions have been implemented. The one that was not implemented is Copy-Paste over a selected range of cells. (It is implemented for a single cell.)

The reason is that the clipboard was storing only one cell value. Extending that to a range would require major changes and may cause inefficiencies.

1.7.2 Challenges Overcome

1) Initially we proposed using `iced` for GUI, but switched to `egui` due to rendering and scrolling issues with `iced`. `egui` is lighter and supports dynamic rendering more efficiently.

2) We faced flickering when changing themes because of interdependent variables. We resolved it by cloning the base variable to avoid conflicts.

3) Rendering the entire spreadsheet was inefficient. We now render only a 300x500 cell window at a time for smoother UI performance.

4) Instead of drag-selecting, range selection is done via right-clicking two opposite corners of a rectangle—simpler and avoids complicated coordinate tracking.

2 Conclusion

Our Rust Spreadsheet is designed to be fast, lightweight, and intuitive. With extensive keyboard support, advanced features, and multiple editing and scrolling modes, it offers a flexible user experience while remaining memory-efficient and performant.