Imperial

SpreadSheets

Java Spreadsheet II

Dependencies and Cycles

COMP40009 – Computing Practical 1

20th – 24th February 2023

1 Aims

- To expand the expression evaluator from part 1 into the core of a graphical spreadsheet application.
- To explore graph concepts such as ordered edges, recursive traversal, depth-first search, and cycles.
- To explore aspects of object-oriented design, such as modelling objects in the problem domain with classes, coupling, and refactoring.

2 The Problem

In part 1 of the spreadsheet task, you implemented the core of a spreadsheet: a parser and evaluator for symbolic expressions, and the ability to store their evaluated values into *cells* which can be referenced by other expressions in other cells. You also implemented a basic text-based user interface to the spreadsheet.

Part 2 requires you to complete the spreadsheet. First, you will integrate the spreadsheet into a graphical user interface. Next, you will implement missing functionality: the ability for changes in cells to propagate through to other cells that reference the changed cell, and dependency cycle detection.

3 What To Do

3.1 A graphical user interface

So far, we have evaluated the cell expression parser and evaluator in the context of a text user interface. Real spreadsheets have a *graphical user interface* (GUI) where the cells are laid out in a two-dimensional table, edited in-place, and recalculations are immediately visible.

For this part of the task, you are given a new skeleton (with new tests). This skeleton contains a basic GUI for the spreadsheet; you do not need to modify the GUI code, unless if you wish to extend it (see section 4). Clone your exercise repository (remembering to replace *login* with your login) with:

```
> git clone
  https://gitlab.doc.ic.ac.uk/lab2223_spring/javaspreadsheetrevisited_login.git
```

You will notice that the skeleton files have been zipped and encrypted with a password. In order to extract them, you will need to use an application that supports 7z files[1]. On the lab machines, this has already been installed for you, so you can extract the archive by typing:

[1] https://www.7-zip.org/download.html

```
7z x skel.7z -p93D57ABE
```

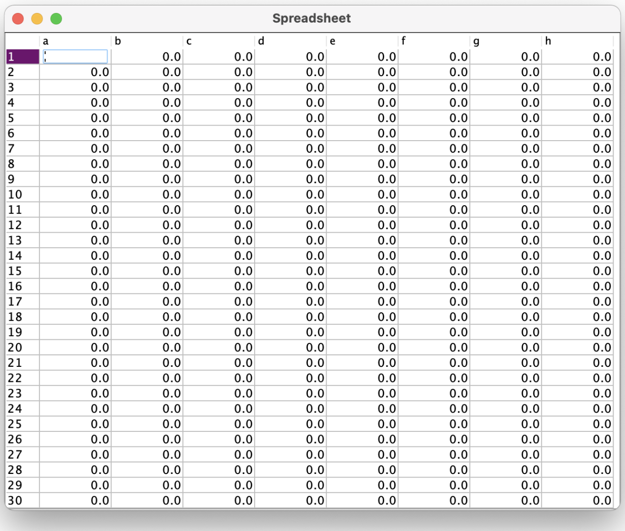
(*n.b., 93D57ABE is the password*). Make sure that you extract the contents of the archive directly inside your repo, i.e. do not create an extra subfolder named "skel", because that would break the LabTS autotesting process. Once this is done, you can then safely delete the 7z archive. Remember to commit and push all the extracted files back into the repo.

You will need to integrate your code from the last part into this skeleton as follows:

- Copy over the `Parser` class and any new classes you made for the first part (such as expression classes). Do *not* copy over your `Main` class; this skeleton uses a different `Main` that runs the GUI.
- Copy over the *body* of your `Spreadsheet` class (constructor, methods, and any fields) from the last part into the skeleton `Spreadsheet` class, replacing the code between the *`// start replacing`* and *`// end replacing`* comments.
- Make sure your code compiles and the tests run (they will fail at first, but should run).

Note that `Spreadsheet` now implements `BasicSpreadsheet` rather than `EvaluationContext`. This new interface contains the various methods needed both for the spreadsheet to interface with the GUI, and for the dependency mechanism we will add later.

Run `Main`; you should see something like the screenshot below. Try double-clicking on cells, typing expressions into them (do not prefix them with `=`), and pressing Enter to calculate the expressions.



Spreadsheet

	a	b	c	d	e	f	g	h
1		0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
7	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
10	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
11	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
12	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
13	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
14	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
15	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
16	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
17	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
18	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
19	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
20	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
21	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
22	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
23	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
24	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
25	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
26	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
27	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
28	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
29	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
30	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

3.2 Storing expressions in cells

You may have noticed when testing the GUI that each cell shows its value initially, goes blank when you edit it, and then shows its new value when you finish editing. In a real spreadsheet, the cell would not go blank, but would instead expand into its last expression, ready for editing.

When editing the cell, the GUI replaces the value with the result of `getCellExpression` for that cell. This method currently returns `""`, but *should* return a string representation of the last expression assigned to the cell. To fix this, and the problem above, we must store that expression somewhere.

We could track the expressions by adding them to the spreadsheet the same way we added values: for instance, if we have a `Map<CellLocation, Double>`, we could add a `Map<CellLocation, Expression>` adjacent to it. This will work, but is not recommended, for several reasons:

- Later on, you may need to associate more state with each cell, and maintaining several maps and methods for each part of a cell will clutter and complicate the `Spreadsheet`;

- A good practice in object-oriented programming is to represent objects in the problem domain with objects in the program. When we talk about spreadsheets, we almost always think of them as containing cells (which, themselves, have values and expressions). Representing this hierarchy in the code will make the correspondence between it and the spreadsheet problem it solves clearer;
- Another good practice is *separation of concerns*: each class should have a well-defined particular responsibility. Keeping all of the logic relating to cells in `Spreadsheet`, without any delegation, means that the spreadsheet has a large number of responsibilities at different levels.

Instead, the skeleton has an empty `Cell` class for you to finish and integrate into the spreadsheet. For now, implement the constructor, `getExpression`, `setExpression`, `recalculate`, `getValue`, `getExpression`, and `toString`. These methods should satisfy the following behaviour:

- New `Cell`s must start in a 'empty' state: with no expression set, and a value of `0.0`. How you represent this state is up to you, as long as empty `Cell`s properly handle the special cases below.
- The `Cell` must store its value separately from the expression. This is for efficiency: while we could calculate the value from the expression every time the value is needed, this would cause large amounts of unnecessary recalculation for cells with many dependencies.
- The `Cell` must recalculate its value from its expression when, and only when, `recalculate` is called. This distinction will help when we introduce dependency management and cycle detection.
- The `Cell` must safely handle attempts to `recalculate` while empty: the value should remain `0.0`.
- `getExpression` must return a `String` that, when parsed, produces an equivalent expression to the one last parsed in `setExpression`. If the cell is empty, it must return the empty string.
- `setExpression` must set the `Cell`'s expression to the parsed form of the string given. If the string is empty, the `Cell` must instead return to the empty state.
- The `Cell`'s `toString` method must return the empty string when the cell is empty, and a string representation of its value otherwise. This reflects the behaviour of 'real world' spreadsheets, where cells only display a value once they have been filled with an expression. It also *decouples* the graphical user interface from any knowledge of the type of value stored in the `Cell`.

3.3 Storing cell objects in the spreadsheet

You should now *refactor* your `Spreadsheet`: alter its internal structure (to use `Cell`s) without changing its external behaviour (the existing interface should not be changed, and the tests should still pass).

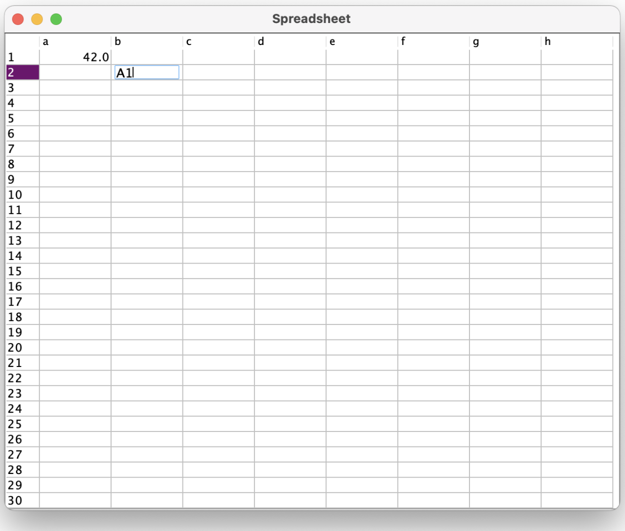
On `Spreadsheet`, change your internal cell storage to hold `Cell`s rather than `Double`s, amending `getCellValue` as needed. Implement `getCellExpression` such that it gets, and `setCellExpression` such that it sets *and recalculates*, the expression of the cell at the given location; if the cell does not exist, it should be created (where appropriate). Finally, implement `getCellDisplay` to return the string representation of the cell. Your spreadsheet should now look like this:

3.4 Dependency management

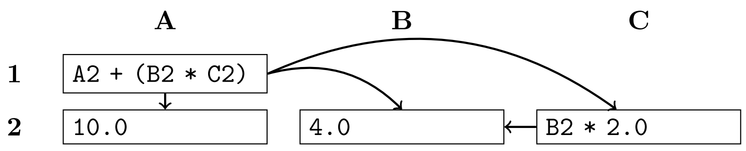
You may have noticed that a key piece of functionality is missing: cells that depend on other cells are not updated if the values of those cells change. Consider what happens if we try to input this spreadsheet:

	A	**B**	**C**
1	`A2 + (B2 * C2)`		
2	`10.0`	`4.0`	`B2 * 2.0`

The value of `A1` should be `42.0`, but, if we fill in its expression before those on row 2, it will be `0.0`. To get the right value, we would need to input `B2` first, then `C2` and `A2`, and then `A1` — even then, any changes to the cells on row 2 will not propagate to `A1`.



To get the correct behaviour, we must take into account the dependency relations between cells. These relations form an *ordered graph*, with edges from each cell that references other cells (the ‘dependent’) to each of the referenced cells (the ‘dependencies’). In the spreadsheet above, for instance, `A1` depends on `A2`, `B2`, and `C2`; conversely, `A2`, `B2`, and `C2` all have `A1` as a dependent. The full dependency graph is:



For changes in cells to propagate correctly according to the dependency graph, each cell must:

- track the other cells that depend on its value (dependents);
- notify dependents when its value has been recalculated;
- ensure that all cells referenced by its expression (dependencies) track the cell as a dependent;
- update the dependency tracking appropriately whenever the cell receives a new expression.

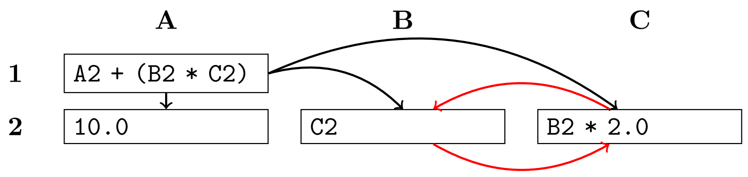
Your next task is to implement this functionality. You will need to:

1. Implement `addDependent` and `removeDependent` in `Cell`. These should track the *dependent* locations of the cell (not its dependencies, which are implicit in its expression).
2. Implement `addCellDependency` and `removeCellDependency` in `Spreadsheet`. These take two locations, a dependency and a dependent, and add or remove a dependency edge between them by delegating to `Cell` where necessary.
3. Change `Cell` such that, when its expression is changed, the cell updates its position in the dependency graph: it will need to remove itself as a dependent from any dependencies it no longer has, and add itself as a dependent to any new dependencies.
4. Implement `recalculate` in `Spreadsheet`; it should trigger a value recalculation on the named cell.
5. Change the cell implementation such that, when we `recalculate` a cell expression, all of that cell’s dependent cells are `recalculated` if necessary. This should then trigger `recalculate` in *those* cells’ dependents, and so on, until all of the cells that depend on the first cell have updated.

Take care to handle dependencies on empty cell locations correctly: if an expression refers to one, it should still be treated as a dependency (and have its dependents tracked), in case it is filled later.

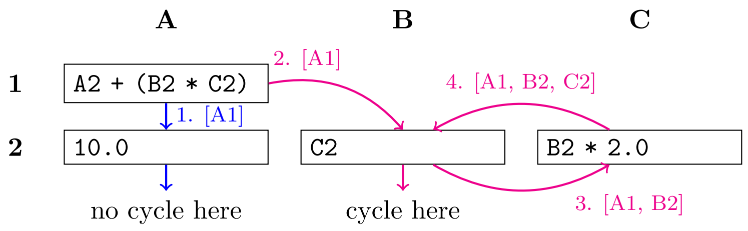
3.5 Cycle detection

Dependency resolution introduces a new problem: if there is a cycle anywhere in the dependency graph, a similar cycle will arise in `recompute` and cause a stack overflow error. Such a cycle can occur, for instance, by trying to set cell `B2` to the expression `C2` in the spreadsheet above, producing the graph:



To prevent this, we must detect and break cycles whenever cell editing introduces them. One way to detect cycles is to traverse each dependency of the cell we just edited (and their dependencies, recursively, in a depth-first manner) and keep track of which cell locations we have seen during each traversal. If at any point the current cell location is one we have already seen, we have detected a cycle.

Here is an example where we perform depth-first cycle detection from `A1`:



The expression in `A1` has three dependencies: `A2`, `B2`, and `C2`. We proceed as follows:

1. We check `A2`, noting we have seen `A1`. As there are no more dependencies, we find no cycle here.
2. We then consider `B2`, again noting we have seen `A1`. `B2` has a dependency on `C2`, so we keep going.
3. We consider `C2`, noting we have seen both `A1` and `B2`. `C2` has a dependency on `B2`, so we keep going.
4. We again consider `B2` but, as we have already seen `B2`, we detect a cycle and finish detection.

The skeleton has an unfinished class called `CycleDetector`: this class has one method, `hasCycleFrom`, that takes a `CellLocation` and returns whether a cycle has been detected *starting from that location*. The detector takes a `BasicSpreadsheet` to find the dependencies of each location. You will need to:

1. Implement `findCellReferences` in both `Cell` and `Spreadsheet`; these should delegate to the cell's expression when one is set.
2. Implement `hasCycleFrom` in `CycleDetector`, as well as that class's constructor if necessary. You may also choose any valid cycle detection algorithm, but should use the `findCellReferences` method of `Spreadsheet` to find each cell's dependencies. Consider the advantages and disadvantages of different approaches.
3. Change `Spreadsheet` so that any change to the expression of a cell will invoke the cycle detector before the cell is recalculated and, if a cycle is detected, do *something* to break the cycle. How you break the cycle is up to you (and perhaps a good point of discussion) so long as the spreadsheet is in a stable state afterwards, and subsequent recalculation of the cell no longer throws an exception.

4 Suggested Extension

Note that, while these extensions will need large-scale changes to the spreadsheet representation, including some of the support code in the skeleton, *the existing tests must still pass without modification.*

- Many spreadsheets allow the insertion of text into cells as well as numbers and formulae; can you implement this on your spreadsheet? (Be careful with the ambiguity between, say, the cell reference expression `A1` and the text 'A1'; consider how real-world spreadsheet programs solve this.)
- Often, when we use spreadsheets, we temporarily introduce non-syntax errors (such as cycles, or divisions by zero) into cells with the intent of resolving them later in the spreadsheet work. Some spreadsheets support this by marking cells with a special *error value* that reports the issue to the user through display strings like `#ERROR` and propagates the error state to dependent calculations. Can you implement this functionality (perhaps using a similar approach to textual values)?
- Extend the GUI code such that it handles the formatting of error and text values differently from numeric values. For instance, many spreadsheets align text to the left, but numbers to the right. You may need to overhaul the way that the GUI receives and stores display information from cells.

Submission

As usual, use `git add`, `git commit` and `git push` to send your created **.java** files to the gitlab server.

Then, log into the gitlab server `https://gitlab.doc.ic.ac.uk`, and click through to your `javaspreadsheetrevisited_<login>` repository. If you view the `Commits` tab, you will see a list of the different versions of your work that you have pushed. Select the right version of your code and ensure it is working before submitting to CATe via LabTS.

Assessment

```
F - E: Very little to no attempt made.
       Submissions that fail to compile cannot score above an E.

D - C: Implementations of most functions attempted;
       solutions may not be correct, or may not have a good style.

B:     Implementations of all functions attempted, and solutions
       are mostly correct. Code style is generally good.

A:     There are no obvious deficiencies in the solution or
       the student's coding style. In addition, there is
       evidence of productive testing.

A*:    As for an A -- plus the student has done additional work
       beyond the basic spec, e.g. by considering (and clearly
       commenting) interesting variations or extensions to the
       given functions; e.g. based on their own research.
```