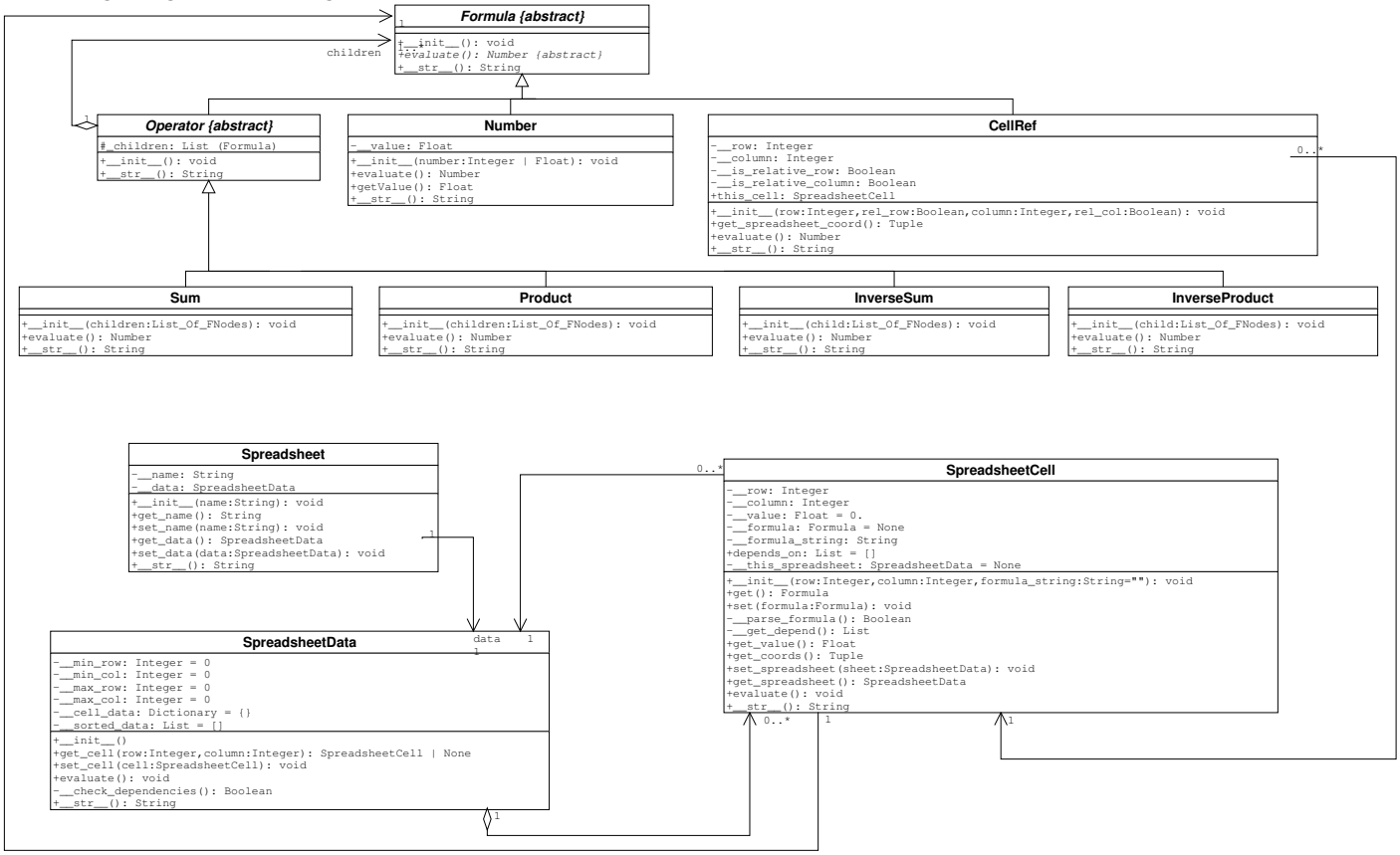
COMP 304B – Object-Oriented Software Design
Assignment 2 – Spreadsheet Design

Solution

The requirements for this assignment are described on a here.

The design is given in the figure below.



The dia file is here. A larger image is here.

Class Diagram

Redesign first assignment

Classes in the `Formula` inheritance tree have only to be modified slightly: the `__str__` method has been added to every class in the inheritance hierarchy.

More importantly, the `CellRef` class has been modified as follows:

- It now has two "`is_relative`" attributes.
- The `this_cell` attribute refers to the `SpreadsheetCell` object the formula belongs to.
- The constructor reflects those changes.

- The method `get_spreadsheet_coord` returns the absolute coordinates (a tuple containing two nonnegative integers) of the `SpreadsheetCell` object the `CellRef` object refers to.

We have three new classes: `Spreadsheet`, `SpreadsheetData` and `SpreadsheetCell`. The first one is self-explanatory.

SpreadsheetCell

The `depends_on` attribute is a list that contains the absolute coordinates (tuple containing two nonnegative integers) of the cells the object depends on.

We added the private attribute `__this_spreadsheet` to refer to the `SpreadsheetData` object the object belongs to. The methods are described below (the methods that were not specified in the requirements are in bold):

`__init__`
- Set the attributes `__row`, `__column` and `__formula_string`.
- Initialize the `formula` attribute with the `__parse_formula()` method.
- Do a DFS on `__formula` to update the `this_cell` attribute in all `CellRef` instances.

`get` Return the contents of the `__formula` attribute

`set`
- Set the attribute `__formula`.
- Do a DFS on `__formula` to update the `this_cell` attribute in all `CellRef` instances.
- Initialize the `__formula_string` attribute by assigning it the string returned by `__formula.__str__()`.

`parse_formula` Parse the attribute `__formula_string` to build the attribute `__formula`. Return `TRUE` only if the parsing is successful.

`get_depend` Do a DFS on `__formula` to update the `depends_on` attribute.

`get_value` Return the content of the `__value` attribute.

`get_coords` Return the tuple `(__row, __column)`.

`set_spreadsheet`
- Set the `__this_spreadsheet` attribute.
- Call the `__get_depend()` method, which updates the `depends_on` attribute.

`get_spreadsheet` Return the `__this_spreadsheet` attribute.

`evaluate` Update the `value` attribute by evaluating the `__formula` (recall: `Formula.evaluate()` returns a `Number` object).

SpreadsheetData

The dictionary of `SpreadsheetCell` references is the attribute `__cell_data`. As explained in the requirements, the entries are indexed by absolute coordinates, *i.e.*, tuples containing two nonnegative integers.

The `__check_dependencies` method (see below) sorts the cells in an order appropriate for efficient calculation. Since a dictionary is not ordered, we need an ordered data structure to hold the sorted coordinates. This is the role of the `__sorted_data` attribute, which is a sorted list of absolute coordinates.

The methods are described below (no new methods were added):

`__init__` The constructor does nothing special.

`get_cell` `get_cell(x, y)` returns the `SpreadsheetCell` object `__cell_data[(x, y)]`, or `None`.

`set_cell` Basically, `set_cell(cell)` registers the `SpreadsheetCell` object `cell` in the current Spreadsheet. Since a `SpreadsheetCell` object knows its coordinates, there is no need to specify `row` and `column` in the method's parameters. If we register a cell in a position where a cell is already present, the latter is just overwritten (disappears thanks to garbage collection).

Pseudo code for `set_cell(cell)`:

- Let `coord = cell.get_coords()`. Register `cell` in the dictionary `__cell_data`, using `coord` as a key.
- Update attributes `__min_row`, `__max_row` `__min_col` and `__max_col` as necessary.
- Call `cell.set_spreadsheet(self)`.
- Call `self.evaluate()`.

`evaluate` Update the whole spreadsheet

- First call the method `__check_dependencies`, which updates the list `__sorted_data` by doing a topological sort of the dependencies.
- If `__check_dependencies` returned `TRUE`, then for each pair `(X, Y)` in the list `sorted_data`, evaluate the corresponding object by doing
  `self.get_cell(X, Y).evaluate()`.

`_check_dependencies` This method does a topological sort of the cell dependencies (updates the `__sorted_data` attribute). For this it needs to look at the `depends_on` attribute of each `SpreadsheetCell` object in the dictionary `__cell_data`.

The method returns `FALSE` *iff* there is a cyclic dependency (including self-loops).

Other Considerations

Initially the `SpreadsheetData` object is empty. There are two ways to create a new `SpreadsheetCell` object:

- *From scratch*: a `SpreadsheetCell` object is instantiated with a formula string,
- *By copying*: a `SpreadsheetCell` object is instantiated without a formula string, and the `set` method is used to specify a formula.

(NOTE: a cell cannot be edited. It can only be overwritten). The method `SpreadsheetData.evaluate` is called every time a new `SpreadsheetCell` object is registered.

Some of the sources of errors are:

- Illegal string in a `SpreadsheetCell` object,
- Reference outside the spreadsheet,
- Dependency cycles.

Below is the pseudo code for the methods `CellRef.evaluate` and `CellRef.get_spreadsheet_coord`: this should explain why a `CellRef` knows its `SpreadsheetCell`, and why a `SpreadsheetCell` knows its `SpreadsheetData` (Note: the method `CellRef.__str__` will be implemented in a similar manner). The code also shows that in case a `SpreadsheetCell` object refers to an empty cell, the latter cell is interpreted as containing the value `0`..

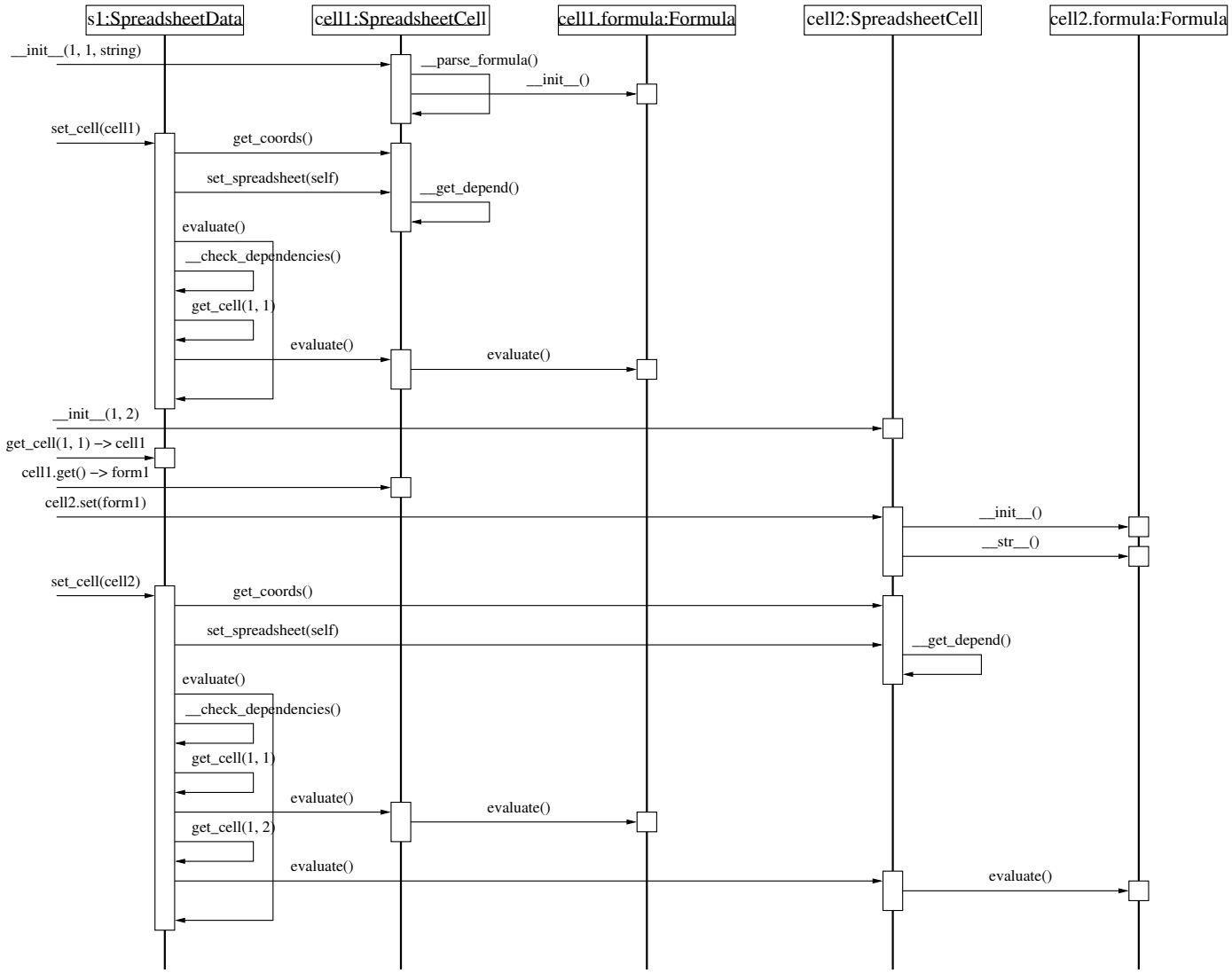
Pseudo code for `CellRef.evaluate`:

- Let `row, column = get_spreadsheet_coord`
  Pseudo code for `CellRef.get_spreadsheet_coord`:
  - Let `cellrow, cellcol = self.this_cell.get_coords()`.
  - If `self.__is_relative_row`, then
    - `row = self.__row + cellrow`
  - else
    - `row = self.__row`
  - (same idea to get `column`)
  - return `(row, column)`
- Let `sheet = self.this_cell.get_spreadsheet()`.
- Let `remote_cell = sheet.get_cell(row, column)`.

- If `remote_cell = None`, then
  - return `Number(0.)`
- else
  - return `Number(remote_cell.get_value()).`
  - (Note that thanks to the topological sort, `remote_cell` is evaluated before the current cell).

Sequence Diagram

In this use case, we assume we start with an empty `SpreadsheetData` object named `s1`.



The first part illustrates how a cell is added by explicitly typing a formula. Let the variable `string` holds the typed formula. We first instantiate a `SpreadsheetCell` object as

```
cell1 = SpreadsheetCell(1, 1, string)
```

In this case, the cell is to be added at coordinates `(1, 1)`. The actual insertion is done by calling

```
s1.set_cell(cell1)
```

The second part illustrates how the cell just added can be copied into coordinates `(1, 2)`. We first instantiate a new `SpreadsheetCell` object without providing a string, and get a reference to the `SpreadsheetCell` object at coordinates `(1, 1)`:

```
cell2 = SpreadsheetCell(1, 2)
```

```
cell1 = s1.get_cell(1, 1)
```
To copy `cell1.__formula` into `cell2.__formula`, we use the `get` and `set` methods:
```
form1 = cell1.get()
cell2.set(form1)
```
Registering the new cell in the spreadsheet is done as before:
```
s1.set_cell(cell2)
```
Note how the whole spreadsheet is updated whenever a new cell is added.