DSheet: The Designed Spreadsheet

Marc Provost
McGill University
marc.provost@mail.mcgill.ca

26th August 2004

Abstract

The DSheet project aims at teaching to undergraduate students how to design a relatively complex application. Most software design examples shown to students come from a wide range of domains and cover only specific problems. The students are never in contact with the design and the construction of a complete application before building it themselves. The main challenge of this project was to find an application that is complex enough to demonstrate software design, while being simple enough to be understood by students without much prior knowledge about design. We think that spreadsheets, which are used by most science students, fulfill this objective. We will show that a spreadsheet can nicely be divided into several small components. Each component can then be designed independently, leading to the iterative implementation of prototypes. It is possible to demonstrate the use of design patterns in the prototypes and the iteration allows the application of regressive testing. This document, which contains the complete design, expressed in UML Class Diagrams, object interaction diagrams and DCharts, was written as a teaching tool for the students. A fully tested, complete and documented implementation of the presented design is also freely available. We think that with DSheet, the students will be able to understand more clearly the links between the various constructs used while designing a complex application.

Acknowledgements

I would like to thank Professor Hans Vangheluwe for his very useful help throughout the project. DSheet would not be as complete without his help.

TODO list

This document contains a lot of information, but is still under construction. If you find that anything is missing or is unclear, let me know and I will add it to the TODO list!

- Prototype 3:
    - Add more information/diagrams about the observer pattern.
    - Discuss the role of the SubjectManager
    - Discuss the implementation in more details.
- Prototype 4:
    - Find a clean approach to automatically test the GUI
    - Describe constraints/actions for all transitions in the statecharts
    - Some design elements are missing, some methods are not described.
    - The final statechart, which handles the timer is not digitalized yet.
    - Add more information about the implementation
- Add used references (Design books, architecture books, etc)
- Improve Conclusion

Contents

List of Figures

List of Algorithms

1 Introduction

This document specifi es in detail the requirements, the design and the implementation of DSheet, a relatively complex spreadsheet application. DSheet was created to demonstrate the software design techniques discussed in `COMP304` ”OO Software Design”. Most of the software design examples that the students encounter during their undergraduate studies are usually simple if not trivial. In contrast, DSheet supports the main features of popular open source (*e.g.,* gnumeric) and commercial (*e.g.,* excel) spreadsheets. A complete design of DSheet, starting from the top-level architecture down to the code itself, is described in this document and is freely available to the students as a learning tool. DSheet is complex enough to demonstrate each step of the software process while being simple enough to be understood by undergraduate students without much prior knowledge about design. The project requirements are specifi ed such that they fulfi ll this general goal. In practice, complex software contains many components, and this project gives an idea of how to build such programs. Components simplify the implementation by dividing the problem into smaller parts. They help the programmers to communicate ideas of a high level of abstraction. They are good for reuse because each of them provides a particular functionality via its interface. DSheet is complex enough to be described by several components. The project also demonstrates how high level design of components is important for reuse, understandability, maintenance, and for shortening the development time. Several design patterns are used and adapted to the components, showing how previous experience accumulated by experts is very useful in new projects. Also, automatic regressive tests are applied when new prototypes are merged with previous ones. Every design decision is thoroughly justifi ed: a comparison of alternatives is documented, since it is important to know why a certain approach was implemented. In the following sections, we will describe the *software process* that will be followed to design and implement DSheet. The remaining sections will present the development of DSheet in the context of this software process.

2 Software Process

As software projects become more and more complex, the importance of building them in a *structured* manner increases to maintain quality. One can easily get confused when dealing with complex problems, which increases the probability of errors. Even if there are no errors in an implementation of a complex problem, the resulting solution could be very inelegant. This leads to software which is long to develop, very hard to understand and, consequently, to maintain and reuse. To reduce development and maintenance costs, developers model how to create a program, from the requirements specifi cation to the implemented solution. This is generally known as the *Software Process*. Several processes were developped and aim to accurately model software creation. In general, developers go through the following steps to create an application:

- Pre-Implementation
  - Problem defi nition
  - Requirements specifi cation
  - Architecture (package-based)
  - Detailed Requirements
  - High-level design (class-based)
- Implementation
  - Low-level design (routine-based)
  - Coding
- Testing
- Deployment, Maintenance

Each flavor of the Software Process spend more or less time on each step. The approach to be used to develop DSheet must have emphasis on design and testing. A *prototype* based software process allows *regressive* testing and solidify the design. Each prototype solves a particular sub-problem of the application and is implemented and tested independently. More complex prototypes are built using previous prototypes. Since each prototype has its own testing suite, all the tests can be applied each time a new prototype is merged with the current application. Such an approach ensures that the design for each component is working since it is implemented independently. Also, prototypes are easier to design since they implement smaller problems that are easier to understand. These steps are followed in the prototype-based approach:

- Pre-Implementation
  - Problem Defi nition
  - High level requirements
  - Architecture (prototype-based)
- Prototype iteration:

  - Subproblem defi nition
  - Detailed requirements
  - High level Design
  - Coding & Merging
  - Regressive Testing
- Deployment, Maintenance

3 Problem Definition

The intention is to build a spreadsheet application that can be used for simple management needs. The hidden objective behind this application is to demonstrate the software process with a focus on design. In the spreadsheet, the user will interact with a grid of cells where data can be inserted. Relatively complex formulas, which can contain references to other cells and functions, will be supported. Undo, redo, copy and paste should be supported in order to show how to design such features. The possibility to have multiple views, potentially on multiple machines, of the same data must be supported. A complete while simple graphical user interface (GUI) must be supported as well as a purely textual interface. The latter will enforce modularity.

4 Requirements specification

The next step in a programming project is to determine *what results are expected* at the end of the development process. There is often, in practice, a mismatch between the *requirements* that are established and the features that an application should have in the future. The apparition of such unexpected features during the development process must be prevented as much as possible because it causes an enormous waste of time and effort. Indeed, such surprises will normally be hard to incorporate in the application since the design was not conceived to handle them. Due to time constraints, one may need to "hack" the feature in the project and doing this will eventually damage the design. This may lead to an application that is no longer modularized, reusable and easy to understand. In the worst case, it could be impossible to implement the feature in the application without rebuilding it from scratch. To avoid the above problems, a fair amount of time will be spent to determine the requirements of DSheet. As one might have expected there is no formal approach to determine the requirements of a project. When humans have no formal science to explain phenomena, they gain understanding by experimentation. But how can one possibly experiment with something that does not exist (yet)? This is a fundamental philosophical question, but note the "yet" in parenthesis. Since one knows that this project will have to exist in the future, one may try to imagine it. A potential approach to generate experience, in a software design context, is to imagine many *use-cases* that the application should support. For each *actor* interacting with the system (*e.g.*,, a client, an employee, a printer, a camera) , describe *how* it interacts via use-cases. Then, from those use-cases, deduce the underlying requirements. This approach is better than simply listing the requirements that one can think of, since it directly brings the context in which the application will be used. Consider the following real experience where that approach was not followed, leading to unpleasant consequences. One of the project requirements was do provide database entries containing information about consumers' habit. Since it was a very large database, the programmer reduced the precision of floating point numbers to economize memory. However, the information was to be used in very complex statistical tests that required very high precision. One can easily imagine the consequences: wrong statistical tests were used for marketing purposes for several months before the error was discovered. And since the error was at a very low level, the whole project had to be reconstructed from scratch (with cost in the millions, due to the incredible amount of information to deal with). If the client had discussed potential use-cases with the developers, the need for floating-point precision would have arisen very quickly. To avoid such problems in DSheet, the use-cases approach will be followed to help determine the requirements, in combination with traditional requirement listing. We combine both approaches because use-cases do not capture everything. There are two kinds of requirements that need to be determined, *functional* and *non-functional*. A functional requirement can be determined by a use-case, because it is a equivalent to a feature of the software. For example, the fact that a cell of the spreadsheet must contain either a formula or a string will easily be captured by a use-case. However, a given use-case will hardly capture non-functional requirements, such as performance or the budget of the project. Those need to be thought of with the client and transformed into a list of requirements. Also, some part of the project, such as the GUI are very hard to describe with use-cases, because they contains a lot of implicit requirements and a given use-case will only capture a small amount of them. They will have to be dealt separately. In the following section, use-cases for DSheet are described and determine its requirements.

4.1 DSheet Use-Cases: Determining the requirements (High Level)

In this section several use-cases that will help determine the requirements for DSheet are presented. The fi rst question that need to be answered is: how many actors are interacting with DSheet? One could say that there is only one actor interacting with DSheet, the user. However, different people use a spreadsheet: scientists, students, professors, managers, etc. What kind of users should DSheet support? DSheet will support data/formula oriented users: it will not have a plotter, solvers or other scientifi c tools. The use-cases will describe how a data/formula oriented user interacts with the system. There are two modes of interaction with DSheet:

- the user is viewing the spreadsheet;

- the user is entering text in a cell (formula or constant data).

There will be one use case for each mode. Each use-case is described by a flow of events.

4.2 Use-Case 1: Viewing a spreadsheet

- *Precondition: The user starts DSheet or ends cell editing*.
- When the user starts DSheet, an *empty spreadsheet* of size `2000` by `2000` is created by default. This size can be modifi ed by the user.
- The user can select cells with the mouse or the keyboard.
- Commands can be applied on selected cells: clear, cut, copy, paste.
- Commands can be undone or redone.
- Commands are can be trigerred with the top menu, the keyboard or the mouse.
- The document can be saved, another document can be loaded.
- Multiple views of the same data can be instantiated.
- New independent views can also be created.
- *Postcondition: The user trigger cell editing with the mouse or the keyboard*

4.3 Use-Case 2: Entering data / formula

- *Precondition: The user triggers cell editing in the currently selected cell with the mouse or the keyboard.*
- If the fi rst character of a cell is “=”, the data is considered to be a formula. Otherwise its constant data.
- Data can be inserted with the keyboard.
- *Postcondition: Cell editing is ended and the data parsed correctly*

4.3.1 What are formulas?

- A formula is an expression consisting of operators and operands that evaluates to a float
- All basic mathematical binary operators are supported: ˆ,+,-,*,/
- Other cells can be operands via cell references. Column are referred as letters, row as numbers.
- It is possible to refer to other cell *relatively* or *absolutely*. The “$” character before a row or a column is used to denote absoluteness. When a formula is moved from cell (x,y) to cell $(x+\Delta x, y+\Delta y)$, the relative rows and columns references are updated respectively by Δx and Δy to reflect the new context.
- Function can also be operands. Functions take an undetermined number of formula arguments.
- The “:” character specify a range of references. Two references, one to the left, the other to the right of “:” denotes the two corners specifying the range.

5 Architecture

This phase of the software process describes, at a very high level, the structure of DSheet. It is particularly important in this stage to work defensively: a mistake in the architecture will affect the detailed design and the actual construction. The more time is invested in this step, the easier will be the implementation. Experimental data shows that a change in the coding requires less time, on average, that a change in the architecture (requirements are even worse, Basili and Perricone 1984). Thus, architecture changes have to be made early to avoid time wastage. Having a well-documented architecture is also desirable because it acts as an abstraction that helps manage complexity: most of the details are handled in the detailed design phase. It can then be used as a means of communication between different agents developing the project: architects, designers, programmers, managers, etc. In fact, each person of the project will refer to it. For instance, the manager will plan his resources by analyzing the architecture, a designer needs to understand the interaction between the component he is trying to design and other components, etc. The architecture should describe in detail what functionality each component provides. Each component can be seen as a collection of classes fulfi lling the same high-level objective. For instance, a high-level objective could be too manage data, parse input or provide a graphical user interface to the user.

6 Detailed Design

An incremental approach is followed to implement DSheet. First, the component having minimal dependencies on other components is designed, tested and implemented. Then, DSheet is incrementally built upon it. In other words, components are topologically sorted by dependencies and they are developed in that order (see Figure 1). Components with a larger set of dependencies reuse the previous prototypes in their testing and implementation phase. This approach allows *regression* testing and solidify the previous developed components. The application grows in complexity as prototypes are extended with more components, ultimately yielding the full DSheet application. In the following sections, this process is explained in detail for each component.

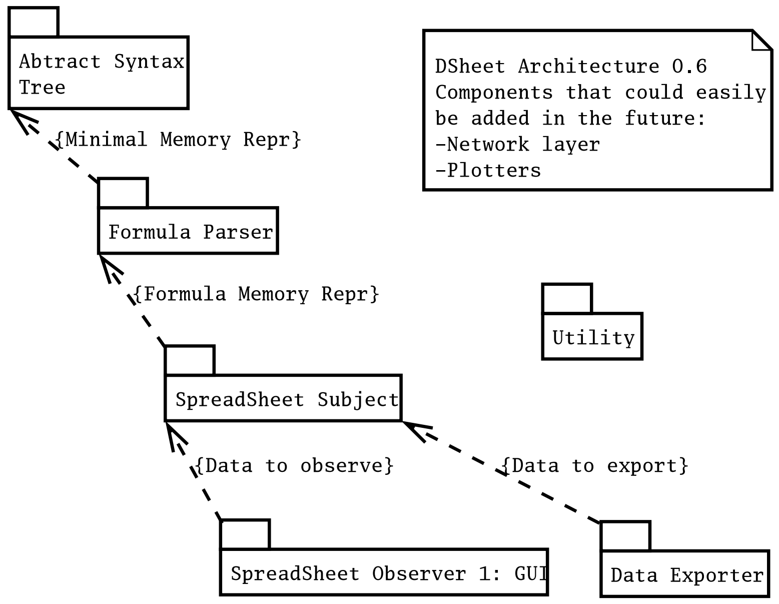


Figure 1: DSheet Architecture, V0.6

Each section is organized as follows:

1. Detailed Requirements Specifi cation (from use-cases)
2. Testing Strategy
3. Design
4. Testing and implementation

For each component, the fi rst step is to determine the *requirements*. This is achieved by analysing the *use-cases* presented in section 4.1 on page 2. A priority scheme will be used for the features that are not implied by a use-case directly. They will be named must-have, good, or optional. They will be considered in the design phase, and their implementation diffi culty should be weighed against their priority. Once the requirements are specifi ed, the testing strategy is established for each requirement and for the component as a whole. Then, the component is designed in detail: the decisions are *justified* carefully, several *versions* of the design are shown and *UML diagrams* are provided for each of them. In short, we go through the thinking process that was taken to achieve the fi nal design of each component rather than to provide the fi nal result of that process. Also, *design patterns* are used wherever they fi t a problem. Finally, when the design is fi nished, the testing suite and the component are implemented. Python was chosen as the implementation language for the fi rst version of DSheet. The scripting language Python has several advantages: quick development, extensive libraries (lists, dictionaries, etc), memory management and object orientation. Furthermore, Python's system is intuitive and results like pseudo-code. This is perfect for a fi rst draft of DSheet. Dynamic typechecking will be manually enforced since Python does not support it. Each time a function is called, its arguments's types will be verifi ed against the expected types. The design presented in this document could be realized in a statically typed and compiled language for effi ciency later. The fi le structure of the implementation of each prototype is organized as follows:

- One directory for each component.
- One fi le for each class of the design
- Within each component, a subdirectory containing the testing suite.

A shell script must be provided to automatically execute the tests. The tests will be implemented using *pyUnit*, python's unit testing framework.

6.1 Prototype 0: Utility module

6.1.1 Problem Defi nition

The utility module will provide global methods and constant values needed by several modules. (*e.g.*, error handling, math functions, conversion functions, etc).

6.1.2 Requirements

This module must support a mechanism for adding features that are needed by several modules. It should be possible to easily add such features in future prototypes. Two features are known to be needed by all the modules: *error handling* and *type checking*.

- Error Handler:
  - A mechanism for handling errors should be supported to prevent data loss after an error (*must-have*)
  - A mechanism for adding features needed by several modules (such as error handling) must be developed.
- Type Checker:
  - The arguments's types of every method must be compared against the expected types. (if not native in the language)

The error handler should handle three level of severity:

- LOG : Display an error msg to stderr (will be used for debugging).
- WARNING : Display an error msg to stderr, but continue execution (*e.g.,* File not found).
- FATAL : Display an error msg to stderr, exit the application cleanly.

Error messages must contain at least the following information:

- Time stamp
- Module name, fi le name, method name, line number
- Severity level
- Description of the error

6.1.3 Testing Strategy

Test methods for success, failure and sanity. Ensure that the errors behave as expected.

6.1.4 Design

The design is based on the *singleton* pattern (see the Class Diagram on Figure 2). There will only be one instance of each `Utility` class at any point in time during execution. As they are needed by future prototypes, global features will be implemented as utilities. In this prototype, two global utilities must be implemented: `ErrorUtility` and `TypeCheckUtility`.

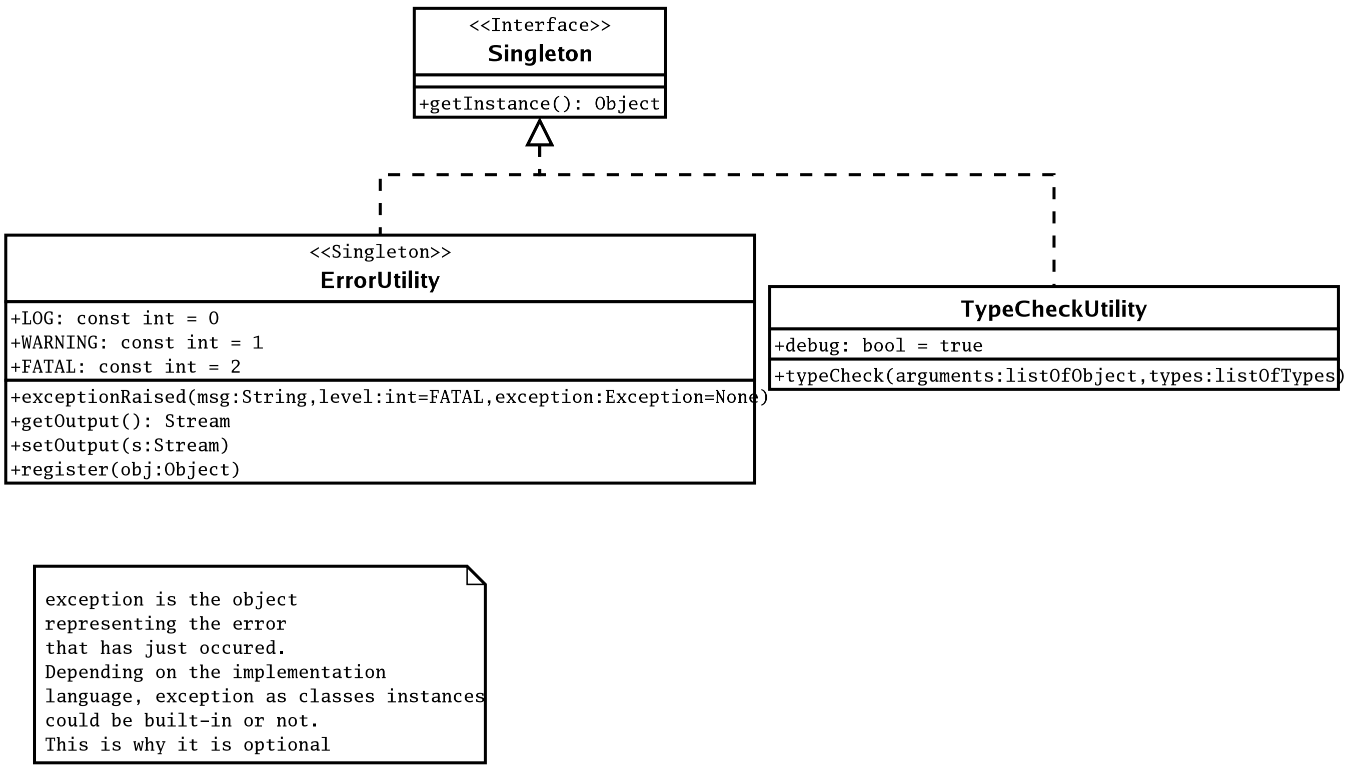
The `ErrorUtility` will be used throughout the application each time the pre-conditions of a method call are not respected. In particular when the types of the arguments are incorrect or when the values of the arguments are not in the expected ranges. Also, the value returned by functions must be verifi ed for correctness when possible. For instance, in the `C` programming language, one should verify that `fopen` does not return `null`. In general, `ErrorUtility` will be used as much as possible to detect errors in the application before they cause bugs that are diffi cult to detect. A class that needs to handle errors will provide three methods: `handleLOG`, `handleWARNING` and `handleFATAL`. At runtime, its instances will register themselves to the `ErrorUtility` singleton, which will notify each registered object every time an error occurs. Registered objects can then react appropriately depending on the level of the error (*i.e.,* save state, display message, etc). `ErrorUtility` also logs every error message to a *stream* (stderr, log.txt, etc) specifi ed at instantiation time. Note that a future sophistication of this design might use a hierarchy of error handlers.

Another utility is used to provide type checking if it is not supported natively by the implementation language. Type checking of arguments is very useful for debugging because it automatically detects trivial misuses of interfaces. The class `TypeCheckUtility` simply verifi es that the arguments of a method correspond to their expected types. If it is not the case, it sends a `FATAL` error message to `ErrorUtility`. A `boolean` flag `debug`, initially set to `true`, enables or disables type checking, which could be disabled for the fi nal release if it causes too much overhead.

It is possible that unforeseen features will need to be globally accessible in the next prototypes. Such features will be implemented in utilities.

6.1.5 Testing and implementation

Since the implementation language does not support type checking, `TypeCheckUtility` must be implemented. Also, `ErrorUtility` must be slightly modifi ed to allow testing for *failure*. `PyUnit` support a mechanism for ensuring that a particular exception has been raised on particular input. This can be used to verify that a feature fails correctly with unexpected input. Since `ErrorUtility` does not explicitly raise exception instances, it is not possible to use `PyUnit` ability to detect errors on invalid input. This can easily be corrected by directly raising the exceptions instead of sending error messages to the registered objects. A boolean flag `isTesting` is setted to `true` when the tests are applied. It would have been possible build our own mechanism for detecting errors by analysing the output stream of `ErrorUtility`, but it is simpler and faster to reuse the existing `PyUnit` approach.



<<Interface>>
Singleton
+getInstance(): Object
<<Singleton>>
ErrorUtility
+LOG: const int = 0
+WARNING: const int = 1
+FATAL: const int = 2
+exceptionRaised(msg:String,level:int=FATAL,exception:Exception=None)
+getOutput(): Stream
+setOutput(s:Stream)
+register(obj:Object)
TypeCheckUtility
+debug: bool = true
+typeCheck(arguments:listOfObject,types:listOfTypes)
exception is the object representing the error that has just occured. Depending on the implementation language, exception as classes instances could be built-in or not. This is why it is optional

Figure 2: Singleton pattern

The tests were implemented in `utility/test/`. They simply verify that `TypeCheckUtility` send a message to `ErrorUtility` when there is a type error, otherwise it should return `true`. `ErrorUtility` is tested by verifying that the correct message is sent when an error occurs. Since this module is very simple, its testing suite is also simple. The testing strategy for more complex tests is explained in the next prototype.

6.2 Prototype 1: Abstract syntax tree

6.2.1 Requirements

The abstract syntax tree (AST) will be the structure representing a formula in memory. It must represent expressions consisting of:

- Operators: +,-,/,*,ˆ
- Operands: References, float numbers, functions, range references
- An example: `(3+4.4)*3-MIN(A1:A3,3.4,B5)*C8`

6.2.2 Testing Strategy

Ensure that, once created, an AST can be traversed properly and that its interface behaves correctly. Again, test its interface for *success*, *failure* and *sanity*.

6.2.3 Design

The design is based on the *composite pattern*. A tree is composed of any number of *nodes*. Each node can either be *terminal* or *non-terminal*. A terminal node possesses no children whereas a non-terminal node is an aggregation of any number of other nodes (either terminal or non-terminal). The general structure of an AST is best described in a Class Diagram, as seen in Figure3.

In order to represent formulae in an AST, several nodes are needed. They can be deduced from the requirements. The terminal nodes can be determined directly from the requirements: numbers, cell references and range references. It is important to understand that cell references nodes, even if they implicitly refer to another cell, are not composite nodes. They will simply hold the column and row value of the reference. This approach is followed because there is no way to resolve a reference when the AST is created: it is not the responsibility of the AST to know how to perform this task. Another component of the design will handle the dependencies between the cells and evaluate them. Also, even though range references (*e.g.*, `A1:A7`) contain two cell references, they are still not composite nodes, since they hold, in place, a fi xed number of specifi c nodes. From the requirements, it is clear that arithmetic expressions

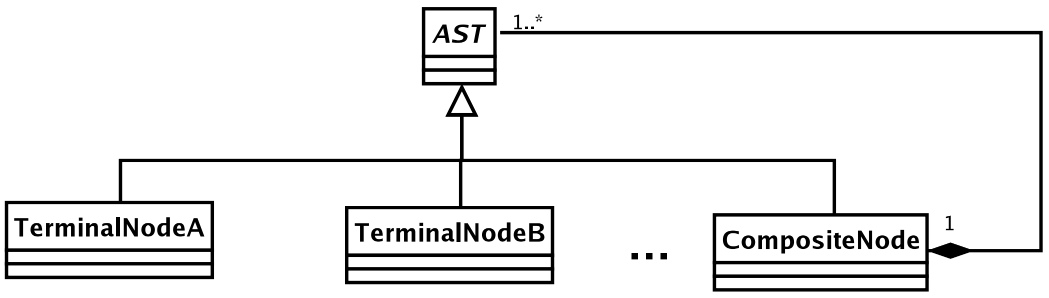


Figure 3: Composite Pattern

such as `4+A1-(3^2+4)+MIN(A1:A7)` and functions such as `MAX(3+3,A1,B1:B6)` are non-terminals. Arithmetic expressions can be subdivided into several composite nodes: a node for sum expressions such as `4+5+A1`, multiply expressions such as `4*3`, inverse sum (or minus) expressions such as `3-4`, unary minus expressions such as`-4`, inverse multiply (or division) expressions such as `4/5` and exponentiation expressions such as `4^5^3`. A fi rst template of the AST design can be seen in the Class Diagram of Figure 4.

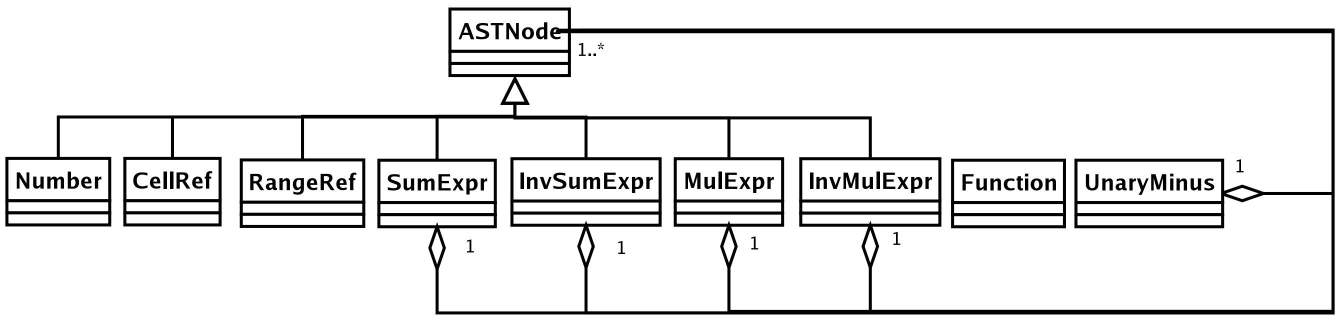


Figure 4: AST Design, fi rst draft

This fi rst draft of the design has some problems. First of all, usually there is only one composite node in the composite pattern. In this case, fi ve nodes are needed and the difference between them is minimal. They all represent the same kind of information and will provide a very similar interface. They will be evaluated using the same algorithm and only the operator will change during the evaluation. There is a very elegant solution to this problem: notice that every composite node can be reduced to a generic `Function` node. For instance, the `SumExpr` node really represents a function called “sum”, the `invSumExpr` node can be replaced by an “invSum” function, etc. This approach has several advantages, the most evident is the simplifi cation of the design, which improves the understandability of this component. The fact that built-in functions (`sum`, `invSum`, `mul`, etc) and optional functions (complex statistical functions, etc) are treated in exactly the same way is a bonus. All the functions will be located in one module and will be easy to test, maintain and modify. Also, reducing the number of nodes will simplify the evaluation process: instead of having to “understand” the `SumExpr` node and evaluate it properly, the evaluator will simply call the function “sum”. The re-factored AST design is shown in the Class Diagram in Figure 5.

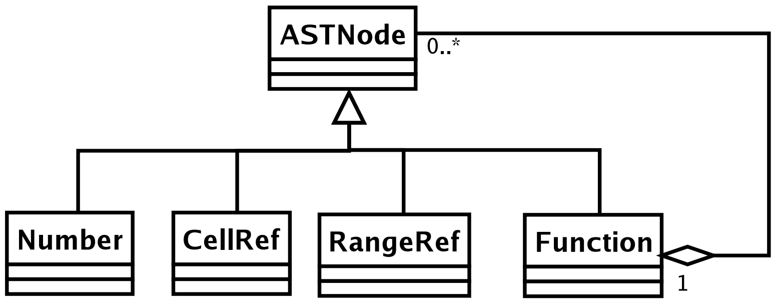


Figure 5: Re-factorized AST Design

The next step is to determine the interface of each AST node. Each node must possess accessors, the question is, should the nodes be mutable or not? More clearly, once the formula is parsed and the tree is constructed, is it useful for an operation to change the value of a given node? For instance, "4+5" is represented by `sum(number(4.0), number(5.0))`. Is it useful to transform the AST to `sum(number(4.0), number(7.0))` in place? We note that, usually, when a formula is modifi ed by the user, it will also be re-parsed because it is impossible to know in advance how much the formula was modifi ed. It could have changed completely. Also, the AST is not modifi ed when it is evaluated (it is simply traversed). It seems that the AST could be immutable, but, to be on the safe side, accessors that modify the state will be provided. It is very probable that an unforeseen feature will require mutability. A preliminary interface is shown in the Class Diagram in Figure 6.

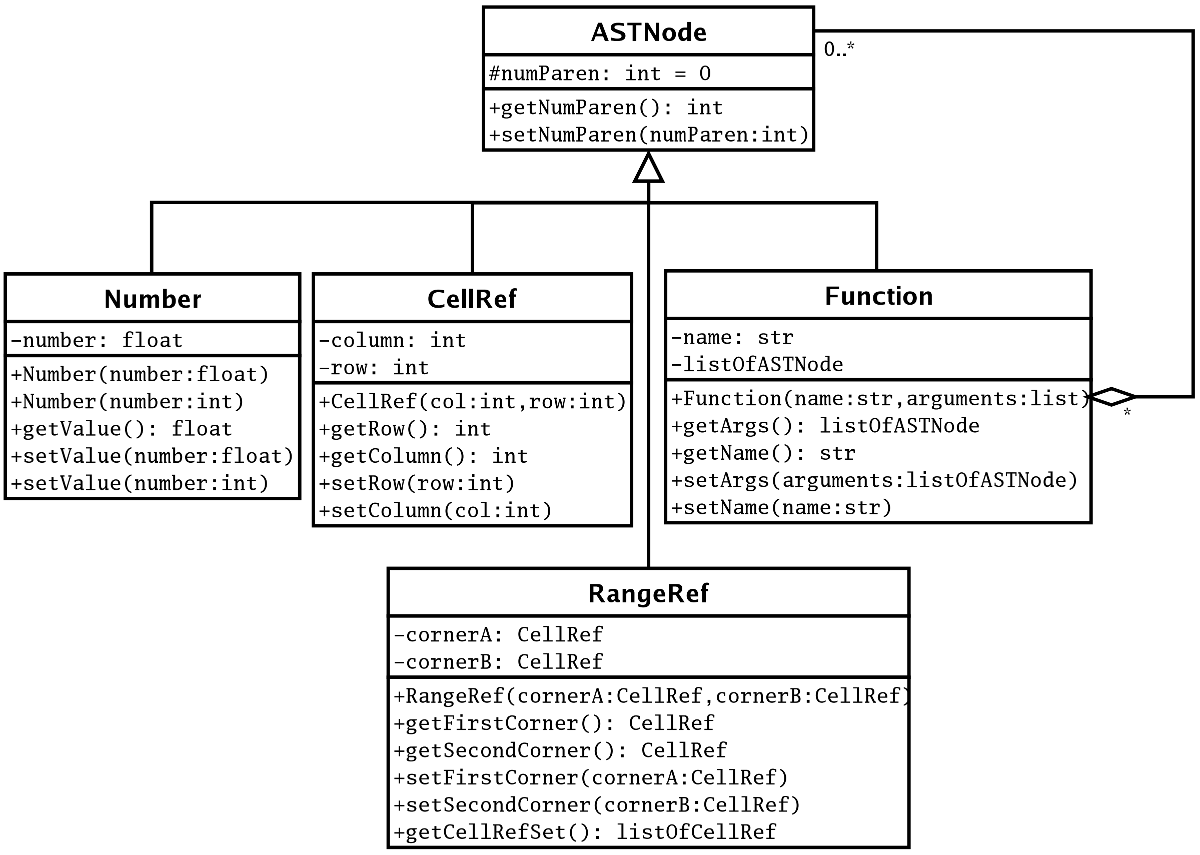
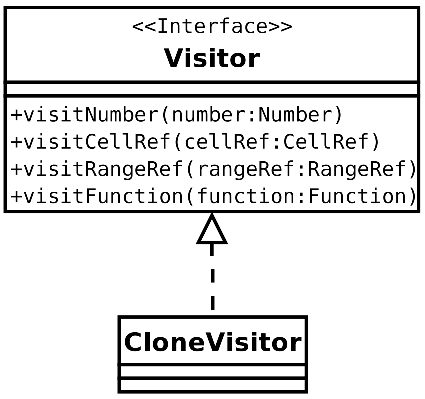


Figure 6: AST Design with Preliminary Interface

Note that a number is always stored as a `float` internally, even though its interface support integers. Integers will be converted to floats. Also, the column value is represented as an `int` for effi ciency, even though it will be displayed as a string (*e.g.,* A5). A converter from integer to the appropriate string representation and its inverse will have to be provided. Why isn't the column value directly stored as a string? Because it is likely that the spreadsheet will be represented, at some level of abstraction, as a two dimensional array. Thus, if the column is stored as a string, it must be converted to an `int` time and again to resolve references. Also, when moving formulas from cell to cell, operations to update relative references will be performed. This will again require a conversion from string to `int`. The only time a column reference needs to be converted to a string is when it is displayed. It is probable that many unforeseen applications will require arithmetic operations on cell coordinates (*e.g.,* in the GUI) and converting from a string to an int in each of these cases would be a complete waste of effort. Thus, we opt for a column stored as an `int` in `CellRef`. These methods will be needed by at least one other module (the GUI needs to diplay columns as strings), so they need to be accessible. It would be possible to declare them class methods in CellRef, which is almost equivalent to declaring them global. These methods are somewhat like math functions, as they do not really belong in the AST module, nor in any other module. Thus, we will opt for defi ning them in as a `Utility` class: `ColumnConvertUtility` shown in the Class Diagram in fi gure XX.

Also, an attribute keeping track of parentheses is added to `ASTNode`: the string representation of the AST will retain *all* the parentheses entered by the user. Of course, it would be possible to compute the *optimal* string representation by removing superfluous parentheses. For example, `(2*3)+3` could be written as `2*3+3`. However, the user may want to keep those parentheses in certain formulas; anticipating later modifi cations, so we will support that. Note that `RangeRef` does not support parentheses (range references are not found in expressions, only in function arguments). An error should be raised if `set/getNumParen` is called by a `RangeRef` object. The `RangeRef` class also has a method that returns a collection of all the cell references contained in the range: this method always returns the cells in a unique order, from top to bottom and from left to right, independently of the range limits.

Some methods need to be added to the previous design, which only describes the *static structure* of the AST. In particular, *operations*, such as `clone`, `equals`, `evaluate` and `toString` could be provided. However, many operations on the AST will be needed by other components in the future: export to permanent storage, copy an AST from a cell to another, compute cell dependencies, etc. It would be very nice to have a mechanism to "objectify" those operations and facilitate their implementation. The *visitor* pattern does exactly this.



<<Interface>>
Visitor
+visitNumber(number:Number)
+visitCellRef(cellRef:CellRef)
+visitRangeRef(rangeRef:RangeRef)
+visitFunction(function:Function)
CloneVisitor

Figure 7: Visitor for the AST

The visitor pattern "visits" each node of the AST and executes an action appropriate for the visited node type. Note how the visitor pattern introduces an `accept` method in `ASTNode` (Figure 8). Each time a node instance of a class `XYZ` is visited, its `accept` method is called by the visitor. This method executes the appropriate action by calling the appropriate `visitXYZ` method in the visitor. To demonstrate this, the Class and Activity Diagram of an objectifi ed "toString" operation are shown in Figures 7, 8 and 9. This is a little exaggerated since "toString" could have been added to ASTNodes interfaces directly without any harm. Still, we opt for this approach to show how to apply the visitor pattern. This is the template for adding other operations on the AST.

In the collaboration diagram shown in Figure 9, it is clear how the operation is performed: the AST is visited in a depth fi rst manner. The terminal nodes are stringifi ed fi rst and the resulting strings are assembled to stringify the composite nodes, eventually yielding the fi nal string representation of the AST. It is in the functions `visitXYZ` that the actual stringifying occurs, thus the implementation of `toString` is external to AST: we have objectifi ed an operation that can be compiled, tested and maintained separately. Note that a method `toString` was added to the abstract class `ASTNode`. This is to simplify the use of the visitor, this method simply creates a visitor, applies it to itself by calling `accept` and returns the result. This method is completely optional since the visitor could be created externally to the `ASTNode` instance. It is there to simplify toString usage. Also note that the string representation for built-in operators $(+, /, , *)$ is handled differently than for normal functions (min,max). The infi x notation is used whenever possible. For instance, `sum(4,5,3)` will always be represented as `4+5+3` (However, `sum(A1:A5)` must be represented with the prefi x notation). It could be possible in the future to add another visitor that provides only prefi x notation, if we fi nd it necessary. The methods clone and equals were added to the design. Clone will mostly be used to preserve encapsulation. Equals returns `true` whenever the AST structure is exactly the same (mostly used in testing). Note that two ASTs evaluating to the same value are not necessary equal (*e.g.*, `AST(2+3).equals( AST((2+3)) )` is `false`).

Another subject that needs to be discussed is the pre-conditions of the interface. For instance, should `CellRef` allow negative values for its rows/columns? Since it is not exactly known in advance what is allowed or not, AST's interface will not be constrained. Constrants will be enforced by the parser, which will produce a valid tree with respect to its grammar.

6.2.4 Testing & Implementation

The testing suites of the fi rst prototype can be found in the directories `../ast/test` and `../utility/test` (for `ColumnConvertUtility`). The suites tests several classes and functions for *success* and *failure*:

- The features that are tested:
  - `Number`
  - `CellRef`
  - `RangeRef`
  - `Function`
  - `ColumnConvertUtility`
- Testing each feature for success, failure and sanity.
  - `Function`: Ensuring that `getArgs` returns a cloned value of its list, not a direct reference. Reason: User should not be able to append to the list directly, but should be forced to use the interface. Note, `getArgs` does not clone the elements of the list (not "deep copy").

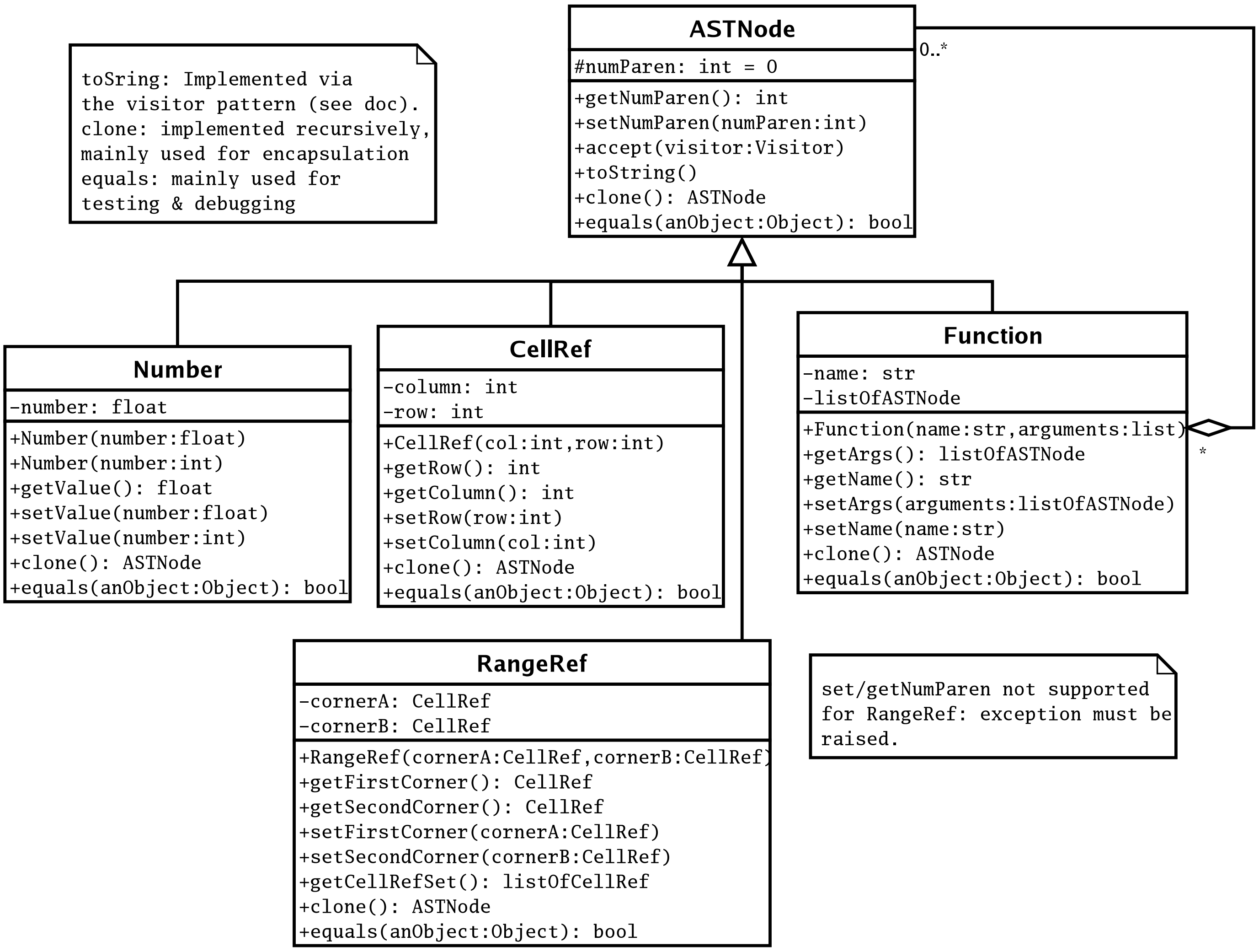

toSring: Implemented via
the visitor pattern (see doc).
clone: implemented recursively,
mainly used for encapsulation
equals: mainly used for
testing & debugging
ASTNode
#numParen: int = 0
+getNumParen(): int
+setNumParen(numParen:int)
+accept(visitor:Visitor)
+toString()
+clone(): ASTNode
+equals(anObject:Object): bool
0..*
*
Number
-number: float
+Number(number:float)
+Number(number:int)
+getValue(): float
+setValue(number:float)
+setValue(number:int)
+clone(): ASTNode
+equals(anObject:Object): bool
CellRef
-column: int
-row: int
+CellRef(col:int,row:int)
+getRow(): int
+getColumn(): int
+setRow(row:int)
+setColumn(col:int)
+clone(): ASTNode
+equals(anObject:Object): bool
Function
-name: str
-listOfASTNode
+Function(name:str,arguments:list)
+getArgs(): listOfASTNode
+getName(): str
+setArgs(arguments:listOfASTNode)
+setName(name:str)
+clone(): ASTNode
+equals(anObject:Object): bool
RangeRef
-cornerA: CellRef
-cornerB: CellRef
+RangeRef(cornerA:CellRef,cornerB:CellRef)
+getFirstCorner(): CellRef
+getSecondCorner(): CellRef
+setFirstCorner(cornerA:CellRef)
+setSecondCorner(cornerB:CellRef)
+getCellRefSet(): listOfCellRef
+clone(): ASTNode
+equals(anObject:Object): bool
set/getNumParen not supported
for RangeRef: exception must be
raised.

Figure 8: AST with updated interface to support the visitor pattern

- Testing that a `TypeError` is raised if an invalid argument is passed to a method.

In the test suite, several sample sets are defined for success and failure. For instance, to test the AST module, we have sets of values that can be used to create `Number`, `CellRef`, `RangeRef` and `Function` objects. Each sample in the set consist of a tuple of elements. The first *n* elements are the inputs needed to test a *feature*, the last element(s) is(are) the expected output. A feature is a set of functions which, given input, provide output. For example, the first sample of the Number set could be (5, 1, "=(5)"). When testing, `Number(5,1)` is instantiated and it is tested for success by verifying that the string representation is "=(5)". In this case, the feature consists of two functions, the constructor and `toString()`. In theory, a set of inputs needs to be provided for every feature. Still, some features are trivial, such as all the accessors (`get/set`). In order to tests them, random (but reproducible) input is generated on the fly.

Also, a set consisting of invalid inputs is provided for each feature that needs to be tested. The testsuite verifies that the expected behavior is observed on bad input (*i.e.,* appropriate exception raised). In particular, the manual typechecking is tested: the testsuite ensures that a `TypeError` is raised when inputs with invalid types are passed to a method.

The implementation is straightforward, but little details need to be discussed. First, `toString` was renamed `__str__` (python automatically calls `__str__` when an object is printed). Also, since Python does not support protected attributes, `numParen` was defined as private and copy/pasted in the children classes along with the accessors.

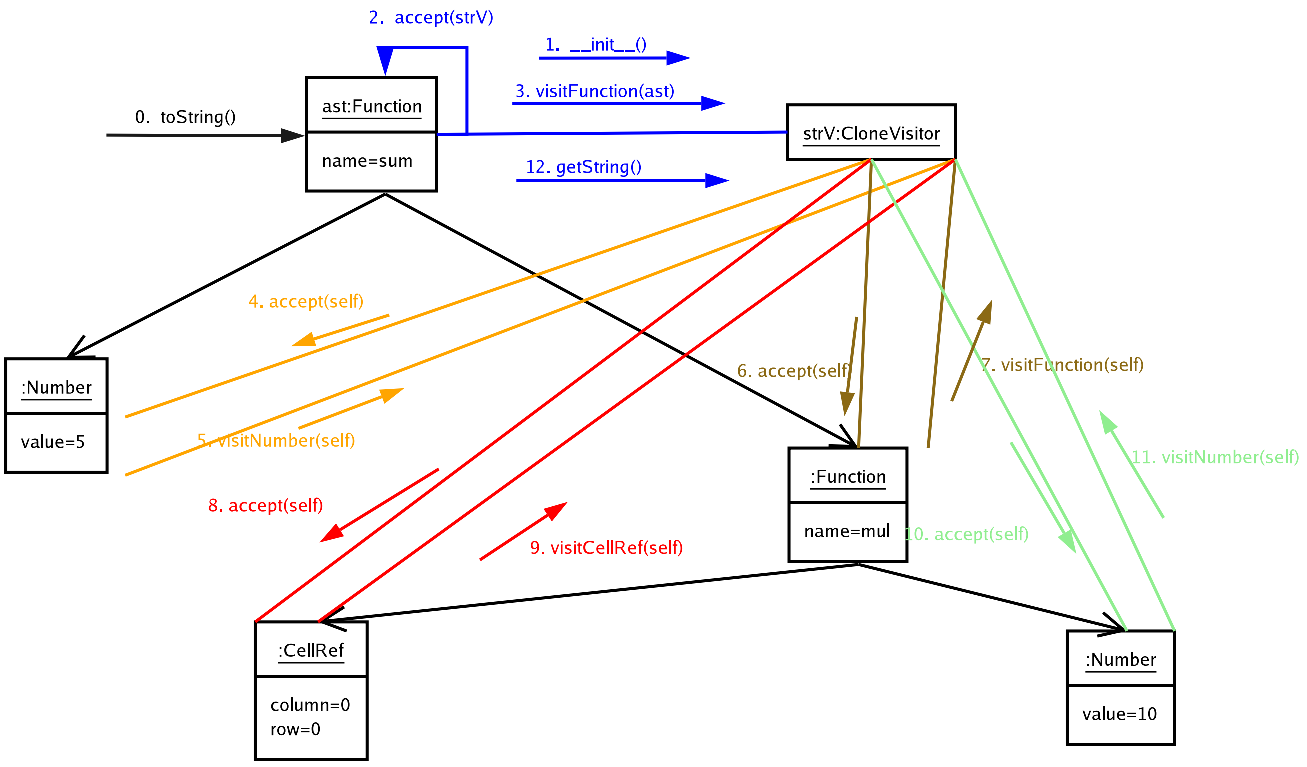


Figure 9: Collaboration Diagram showing the execution order of the `toString` operation on the formula `sum(5,mul(A1,10))`

6.3 Prototype 2: Formula Parser

6.3.1 Problem Definition

In this prototype, we add the formula parser, which will interpret formula strings and generate an AST from it. This module will be used by the spreadsheet data subject in order to “understand” formulas and evaluate them.

6.3.2 Detailed Requirements

Cell References:

- A base-26 system is used for the cell references. The first 26 columns are marked with A-Z, Then, the next 26 columns are marked AA-AZ, etc. (*must-have*)
- References are uppercase internally (simpler). (*must-have*)
- Convert lower case references to upper case automatically for the user. (*good*)

Functions:

- Functions are not case sensitive, which is easier for the user. (*good*)
- Functions must be very easy to add by the programmer. (*must-have*)
- Function definitions can be added by the user statically (*e.g.*, within a file). (*good*)
- Names of the functions should be small, which is easier to use. (*must-have*)
- Minimum set of functions: min,max,avg,med. (*must-have*)
- Aliases could be provided, such as minimum=min. (*good*)

General:

- A *grammar* is needed to formally describe a language for formulas.
- Functions can be used as operands and can be nested inside other functions. (*must-have*)
- Functions have an unbounded number of arguments. (*must-have*)
- Range references cannot stand by themselves. They must be an argument of a function. (*must-have*)

- To support the 25000 cells requirement, 4 digits are allowed for the rows (up to 9999) and 3 letters are allowed for the columns (up to 17576). Must be easy to change (*e.g.*, via confi guration fi le, make). (*must-have*)

We now need to formally specify DSheet formulas. Traditionally, a *scanner* is the module which will recognise the *words* of a language. For example, “for”, “while”, “int”, “float”, “struct” would be valid words recognised by a scanner for the programming language `C`. To formally specify what tokens or words are accepted by a scanner (*i.e.*, the requirements of a scanner), the regular expression notation is used. The syntax of a given regular expression uses a few simple operations on the characters of an alphabet:

a	an ordinary character stands for itself
ε	The empty string
	another way to write empty string
$M \vert N$	Alternatives, M or N
$M \cdot N$	Concatenation, M followed by N
MN	Another way to write concatenation
M^*	Repetition, zero or more times
M^+	Repetition, one or more times
$M?$	Optional, zero or one occurrence
$[a-zA-z]$	Alternatives in a set of characters, calculates a contiguous range
.	A period stands for any single character except newline
“$a.+*$”	String in quotes stands for itself literally

For example, valid expressions could be

“while”	Represent the string “while”
$[a-z][a-z0-9]*$	Represent legal variable names in a programming language
$[0-9]+$	Represent an integer number
$([0-9]+"."[0-9]*) \vert ([0-9]*"."[0-9]+)$	Represent a floating point number

Thus, a regular expression will have to be specifi ed for each legal token that can be in a cell formula. First, a representation for a number, which is either an integer or a real, is needed. The legal characters will be letters. The legal strings will be a combination of characters and integers. A reference to another cell is simply a combination of letters, numbers and $. Finally, binary operators such as “+” are also legal tokens. The scanner specifi cation for the formulas is summarized in the following table:

int	$[0-9]^+$
real	$([0-9]^+ "." [0-9]^*) \vert ((([0-9]^* "." [0-9]^+)$
number	int $\vert$ real
char	$[a-zA-Z]$
string	$[0-9]^*$char$^+$(char$\vert$$[0-9]$)*
ref	[$]?[A-Z]·[A-Z]?·[A-Z]?[$]?[1-9][0-9]?[0-9]?[0-9]?
rangeref	ref ’:’ ref
addop	$[+-]$
mulop	$[*/]$
expop	[ˆ]
funcname	char$^+$
lpar	“(”
rpar	“)”
comma	“,”
equal	“=”
text	All strings that do not start by ‘=’

The scanner will break an arbitrary stream of characters into a stream of individual *legal* tokens. When trying to match a token, the scanner will always return the *longest* possible match. For instance, if it receives the input `1123.235`, it will not stop at `1` and return an `int` token, but will continue to match the longest token, which will be a `real` in this case. Now, requirements that defi ne legal words in a cell have been settled, but another module is needed to verify that the tokens are in the correct order. Traditionally, this module is called a *parser* and its requirements are specifi ed via *grammars*. A grammar, G, is a structure $\langle N, T, P, S \rangle$ where N is a set of *non-terminals*, T is a set of *terminals*, P is a set of *productions*, and S is a special non-terminal called the start symbol of the grammar (In this document, the first non-terminal of a grammar is considered to be S). For example, the following grammar recognise simple addition expressions (*e.g.,* `4+5+4+832;`):

$$\begin{array}{rcl} PROG & \rightarrow & EXP\,; \\ EXP & \rightarrow & EXP\; addop\; EXP \mid int \end{array}$$

In the example, non-terminals are written in uppercase while terminals are written in lowercase (grammars will always be described like this in this document). There must a production for each non-terminal, which evaluates to a terminal at some point (to avoid infinite recursion). One can also notice the link between a parser grammar and the corresponding scanner: the tokens produced by the scanner correspond to the terminals of the parser's grammar. In order to *parse* a stream of tokens, the parser will query the scanner for the next token and verify whether it is legal with respect to the current production. For instance, in the previous example, the input `4+5+a` would fail, because the token `a`, which is a `char` does not correspond to a terminal in the production `EXP` (the only terminal is `int`). Next follow the grammar for the parser of DSheet:

$$\begin{array}{rcl} FORMULA & \rightarrow & equal\; EXP \\ EXP & \rightarrow & EXP\; addop\; TERM \mid TERM \\ TERM & \rightarrow & TERM\; mulop\; EXPONENT \mid EXPONENT \\ EXPONENT & \rightarrow & EXPONENT\; expop\; SIGNEDFACTOR \mid SIGNEDFACTOR \\ SIGNEDFACTOR & \rightarrow & '-'\; FACTOR \mid FACTOR \\ FACTOR & \rightarrow & lpar\; EXP\; rpar \mid number \mid FUNCTION \mid ref \\ FUNCTION & \rightarrow & funcname\; lpar\; (ARG \mid \varepsilon)\; rpar \\ ARG & \rightarrow & (rangeref \mid EXP)\; ENDARG \\ ENDARG & \rightarrow & comma\; ARG \mid \varepsilon \end{array}$$

6.3.3 Testing Strategy

The parser should be tested as follows:

- For success: assert that a given formula produce the expected tree
- Failure: assert that invalid formulas are not parsed and that an error message is produced
- Sanity: ensure that the string representation of an AST is equal to the parsed string. Also, assert that, when re-parsed, the string representation of an AST yields the same AST.

6.3.4 Design

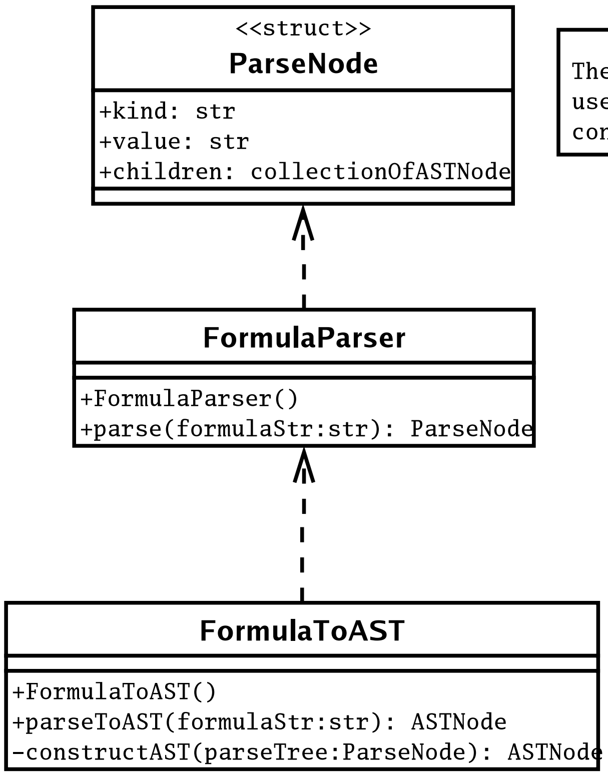
The actual parser will be generated automatically by a parser generator given the grammar specification. Since parser generation is implementation specific, it will not be discussed in detail here. Please refer to the implementation section for a detailed example.

Once the parser is generated, a `FormulaParser` class, with a parse method, should be provided to encapsulate the automatically generated parser. This class will produce a parse tree: a non-optimal tree containing all the grammar artifacts. This tree is an aggregation of *parse nodes*, each labelled with a given grammar (non-)terminal. The parse tree can be seen as a a record of the grammar rules that were applied when parsing. A parse node with a label “X” is created each time a (non-)terminal “X” is encountered. When the parser reaches a terminal, the variable “value” of the parse node is used to store the terminal. The parse node could be implemented as a struct in C or as a class with public attributes in any OO language. The parse tree needs to be simplified into an Abstract Syntax Tree where all redundant information has been stripped, making it easier to process. As seen in Figure 11, the tree will be reduced by removing all the non-terminal nodes and replacing them by Function nodes.

As seen on the UML diagram of Figure 10, to parse a formula, one simply needs to call “formulaToAST”. This method will call the parser and transform the retrieved parse tree into an AST (via the private method constructAST).

6.3.5 Testing & Implementation

In this module, only one feature needs to be tested: the parser. In order to test if the AST structure that was generated by the parser is correct, a simple evaluator visitor was written. This visitor evaluates simple expressions without cell references. The first test parses such expressions and verifies that each of them evaluates to the correct value. In the second test, the expected AST structure was manually constructed. The test simply ensures that the tree that is returned by the parser is equal to the expected AST. The next tests



<<struct>>
ParseNode
+kind: str
+value: str
+children: collectionOfASTNode
The stuct stereotype is used to denote a class containing only attributes
FormulaParser
+FormulaParser()
+parse(formulaStr:str): ParseNode
FormulaToAST
+FormulaToAST()
+parseToAST(formulaStr:str): ASTNode
-constructAST(parseTree:ParseNode): ASTNode

Figure 10: UML Class diagram for the formula parser

uses the built-in string representation (equivalent to pretty-printing). It ensures that the string representation of the returned AST is equal to the parsed string. A sanity test is also applied: if re-parsed, the string representation should return an AST which is equal to the original AST. Finally, the parser is tested for failure: invalid formulas are parsed and the testsuite verify that the parser returns an error. Please see the directory `formulaparser/test/` for more details.

YAPPS, Yet Another Python Parser System, was used to generate the formula parser. This system is very simple and generates human readable Python output. YAPPS produces *recursive descent* parsers, so the LALR(1) grammar that was specifi ed in the requirements was converted to a LL(1) grammar. In a recursive descent parser, each non-terminal is mapped to a function. In order to parse, each function recursively calls other functions representing the non-terminals in the rule specifi cation. For instance, $EXP \rightarrow TERM\ add\ TERM$ could be encoded as:

```
def EXP():
  TERM()
  scan("add")
  TERM()
```

A recursive descent parser cannot handle one or more rules having themselves as the fi rst non-terminal of their defi nitions. For instance, a rule like $EXP \rightarrow EXP\ add\ TERM$ would cause infi nite recursion (this problem is called *left recursion*). Our grammar was transformed into one without such rules, a LL(1) grammar. To achieve this, we replaced left-recursion by right-recursion and ensured that all the rules can be identifi ed with one look-ahead token. For instance, the fi rst rule was converted like this:

$$EXP \rightarrow EXP\ addop\ TERM \mid TERM$$

$$\begin{aligned} EXP &\rightarrow TERM\ RESTTERM \\ RESTTERM &\rightarrow addop\ TERM\ RESTTERM \mid \varepsilon \end{aligned}$$

YAPPS supports regular expression-like syntax to reduce the number of rules in the LL(1) grammar. It is possible to use the “*” or “+” notation to describe lists of (non-)terminals. With this notation, EXP and RESTTERM rules can be merged:

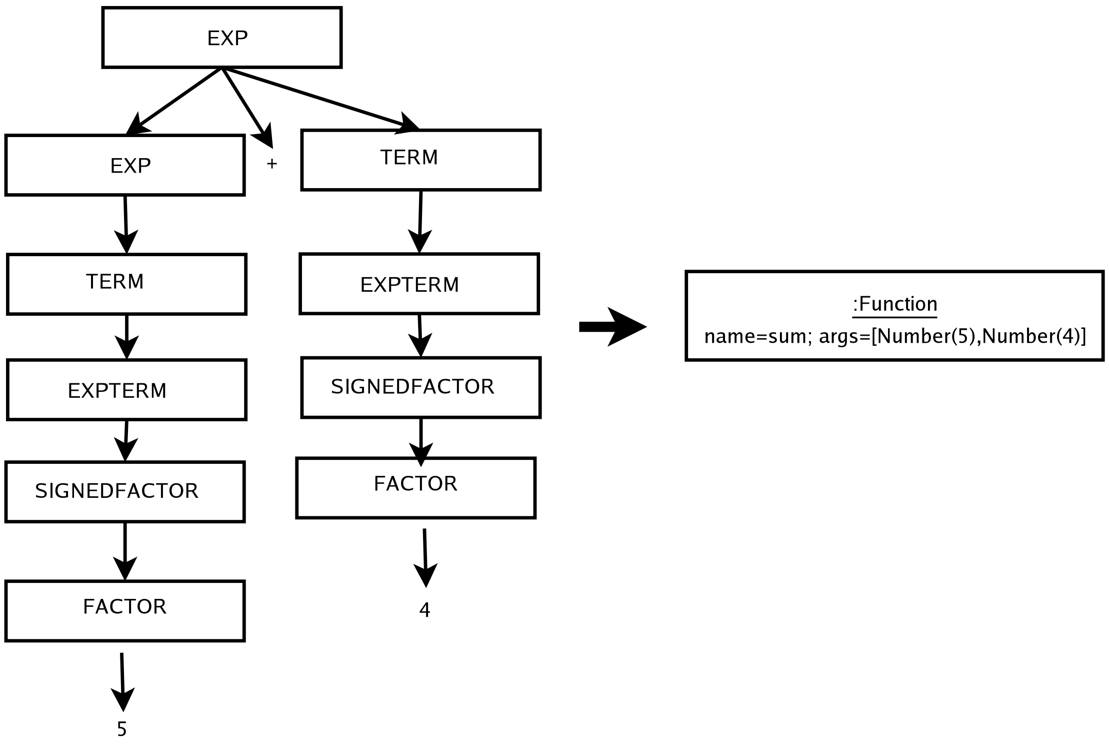


Figure 11: A sample parse tree and its reduced AST form for the formula `5+4`

$$EXP \quad \rightarrow \quad TERM\ (addop\ TERM)*$$

Applying this technique for every rule yields the final LL(1) grammar:

$$\begin{aligned}
EXP &\rightarrow TERM\ (addop\ TERM)* \\
TERM &\rightarrow EXPTERM\ (mulop\ EXPTERM)* \\
EXPTERM &\rightarrow SIGNEDFACTOR\ (expop\ SIGNEDFACTOR)* \\
SIGNEDFACTOR &\rightarrow FACTOR \mid sub\ FACTOR \\
FACTOR &\rightarrow lpar\ EXP\ rpar \mid number \mid FUNCTION \\
FUNCTION &\rightarrow funcname\ lpar\ ARG\ rpar \mid funcname\ lpar\ rpar \\
ARG &\rightarrow (rangeref \mid EXP)\ (comma\ (rangeref \mid EXP))* \\
FORMULA &\rightarrow equal\ EXP
\end{aligned}$$

The specification of this grammar for YAPPS can be found in the file `FormulaParser.g`. All capitalized and lowercase artifacts are non-terminals and terminals, respectively. Note that there are slight discrepancies with respect to the design. For instance, `addop` was divided into two separate operators, `add` and `sub` to perform the correct actions with YAPPS: choose between creating a `sum` or an `invSum` parse node. A similar transformation was performed on `mulop`. The implementation was generated automatically by YAPPS and can be found in the file `FormulaParser.py`.

6.4 Prototype 3: DataSubject

6.4.1 Problem Definition

In this prototype, the module for handling data is added: the datastructure for representing the spreadsheet will be implemented. Once it is completed, it will be possible to create spreadsheets and modify their contents. Note that there is still no GUI

6.4.2 Detailed Requirements

General:

- formulas must be evaluated. (*must-have*)
- cycle of references must be detected. (*must-have*)
- it must be possible to set/get/delete the data contained n one or more cells. (*must-have*)

Performance:

- Theoretically, there are no spreadsheet size limit. In practice, interaction with the spreadsheet must be real-time when the total number of cells is smaller than one million. (*must-have*)

Commands:

- unlimited levels of undo/redo. (*must-have*)
- copy/paste. (*must-have*)
- insert cells/rows/columns. (*optional*)
- select a row/column or all cells. (*good*)
- search/replace (*optional*)

Evaluation Errors:

- Special values indicating the kind of error must be displayed in the cell(s) containing the error. (*must-have*)
- Only the cells that needs to be edited in order to fi x the error should contain a special value. (*must-have*)
- Display the error flag in the part of the formula where the error occurs, to minimize correction work. (*good*)
- Empty cells are considered as `0` in the evaluation mechanism. (*must-have*)
- Cycles should not block the evaluation of cells outside the cycle. (*must-have*)

6.4.3 Testing Strategy

This module is signifi cally larger than the previous ones. Classes will be topologically sorted by dependencies and tested in that order for *success* and *failure*. *Sanity* tests will be applied where possible.

6.4.4 Design

The fi rst class that is needed in the `DataSubject` module is one which encapsulates the data and provides an *interface* for interacting with it. `SSheetData` will encapsulate an $m \times n$ chart of `SSheetCell` instances. The class `SSheetCell` will hold the data entered in individual cells. Two types of data can be entered: a formula or a string (a string does not start with “=”). If a formula is entered, it must be parsed into an AST and evaluated to a float value. Otherwise, the string is trivially “evaluated” to a string value. In summary, `SSheetCell` contains the following information:

Field	Type	Default Value	Description
formula	string	empty	Hold the formula string
ast	ASTNode	null	Hold the AST representation of a formula
value	string or float (union)	float: 0.0 string: empty	If there is a formula: Hold the evaluated float value Otherwise: Hold a string value (non-formula data)

None of the three fi eld is mandatory. Depending on the situation, one or more fi elds will be used:

Data entered	formula field	ast field	value field
a `formula` (*e.g.*, `=3+4`)	used	created using the parser	created using the cell evaluator (float)
a `string` (*e.g.*, `marc`)	not used	not used	used (string)
an `ast`	created using `ast.toString()`	used	created using the cell evaluator (float)

Thus, a `formula` is parsed within `SSheetCell`. Note that reparsing can be avoided if an `ast` is used directly. This approach will be used to update cell references while pasting (explained later). The UML diagram in Figure 12 shows a preliminary interface for the `DataSubject`. Note that the class `SSheetState` is used to modify and retrieve the state of `SSheetData`. It is an independent *layer* between the `DataSubject` and its *observers*. It hides the internals of the `DataSubject` because it contains only the information used to create `SSheetCell` instances. This is done mainly to avoid parsing the formulas outside of the `DataSubject` (this reduces the dependencies between the modules).

The choice of the datastructure to be used in `SSheetData` is very important. A good choice imply good performance and ease of maintenance. Next follows a chart summarizing the performance of possible datastructures. Note: the word *cell* is used to denote a `SSheetCell` instance. The following factors are used to compare them:

- Adding a cell to the datastructure
- Retrieving a cell from the datastructure
- Deleting a cell from the datastructure
- Retrieving a range of cells from the datastructure. (*i.e.*, all the cell in between the cells (x,y) and (x+k,y+u))
- Resize needed? Does it need to be completely copied inside a larger memory location sometimes?
- Memory Usage

DataStructure	Addition	Retrieval	Deletion	Range Retrieval	Resize?	Memory
Array	O(1)	O(1)	O(1)	O(1)	Yes (outside scope)	Maximal
Sorted Linked List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	No	Minimal
Hash Table	O(1)	O(1)	O(1)	O(n)	Yes (Fixed Num keys)	Medium
Hash Table (with sorted keys)	O(log(n))	O(1)	O(log(n))	O(log(n))	Yes (Fixed Num Keys)	Medium

From the data of the previous chart, we deduce that an *array* of references to cells is the best solution with respect to operations on the cells. Its main disadvantage is that it needs a lot of memory because it must be resized when a cell is added outside of its scope. Thus, performance can be dramatically affected when an array is used and the cells are spreaded. A solution to solve the resize problem is to use a *linked list*. This approach has no resize and uses a minimal amount of memory, but its performance is average for every operation. The *hashtable* support spreaded cells and maintains good performance for the operations. Indeed, its resize is independent of the keys used to map the cells. However, this solution comes with a price: range requests on hashtables are executed in linear time instead of constant time. Assuming that range requests are signifi cally rarer than single requests, this solution is good. Range requests can be improved if the keys of the hashtable are sorted. This maintains constant retrieval time, but worsens addition and deletion. The choice of the datastructure is based on usage assumptions:

Data Quantity	Data Behavior	Range Requests	Solution
High	–	Common	Array
High	–	Rare	Array or Hashtable
Medium	Dense	Common	Array
Medium	Dense	Rare	Array or Hashtable
Medium	Spreaded	Common	Array or Hashtable
Medium	Spreaded	Rare	Hashtable
Low	–	–	Linked List

Overall, the hashtable behaves better than other datastructures. We will opt for this datastructure in `SSheetData`. The hashtable will map a CellCoord object, which simply encapsulates a *column* and a *row*, to a `SSheetCell` instance. The `CellCoord` object must provide a `hash` function that will be used by the hashtable. `SSheetState` will also use a hashtable mapping a `CellCoord` instance to a tuple containing the cell value, the formula string and the ast of a given cell. Note that a CellCoord that is not mapped in the hashtable represent an empty cell. Thus, `SSheetState` will return default empty values when it is queried with an unmapped `CellCoord`. For effi ciency, `SSheetData.setState` takes a state which contains only the modifi ed cells. The following chart summarize the action performed by setState depending on the input:

CellCoord exists?	Formula String	AST	CellValue	Action Performed
no	empty	null	null	Do nothing
yes	empty	null	null	Remove the cell
yes	non-empty	null	—	Set the formula, parse and evaluate
yes	—	non-null	—	Set the formula (str(ast)) and evaluate
yes	empty	null	CellValue(stringValue=non-empty)	Set the value and evaluate

The next feature that need to be designed is the formula evaluator. This evaluator will *interpret* each `ast` of `SSheetData` by computing the mathematical result of the corresponding formula. Since a formula can refer to other cells (*e.g.*, `=4+A3`), the evaluator will need a reference to `SSheetData` to evaluate it. The naive approach to resolve cell references is to query `SSheetData` in an undefi ned order. For instance suppose we have the following spreadsheet:

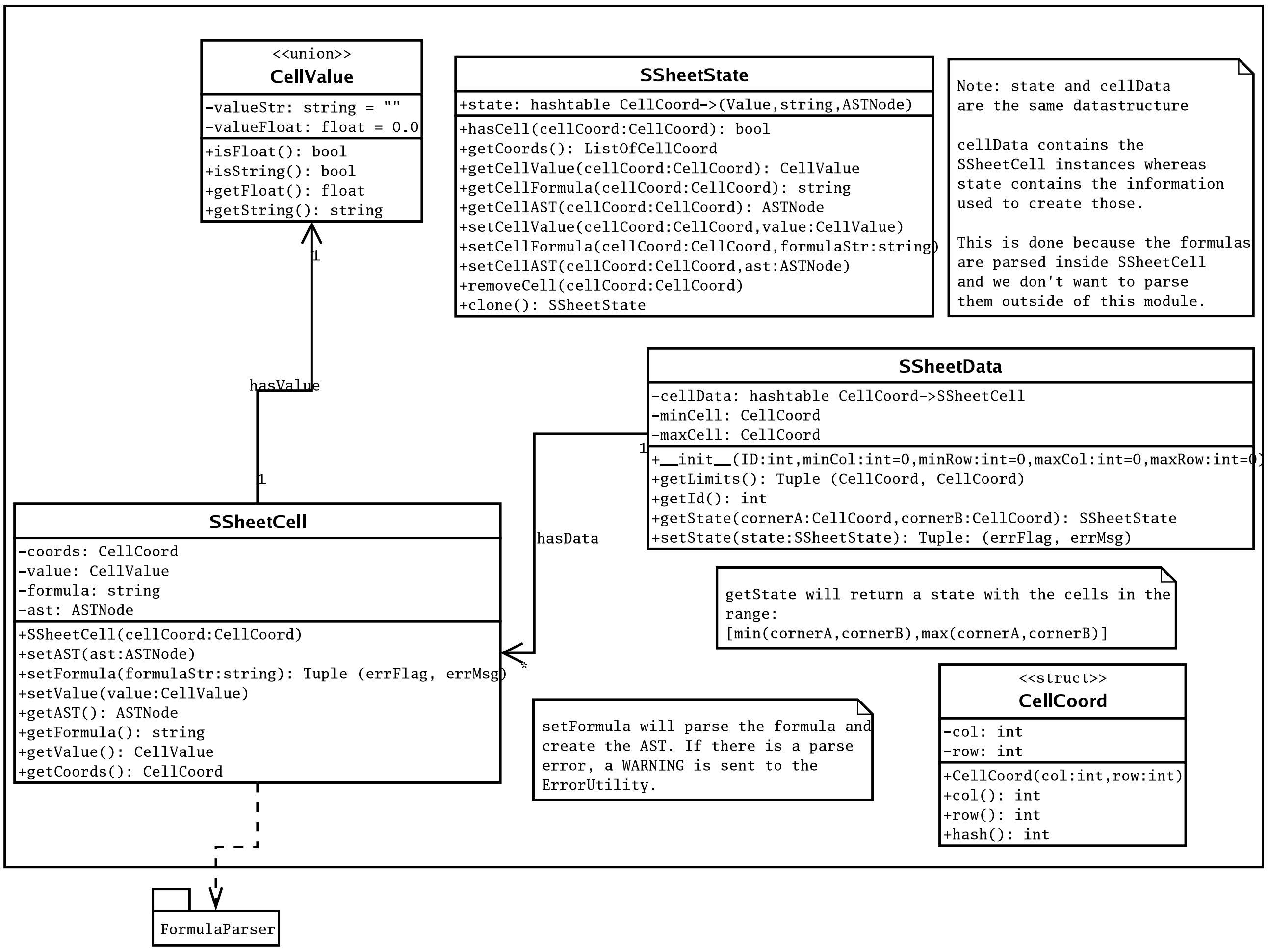

<<union>>
CellValue
-valueStr: string = ""
-valueFloat: float = 0.0
+isFloat(): bool
+isString(): bool
+getFloat(): float
+getString(): string
SSheetState
+state: hashtable CellCoord->(Value,string,ASTNode)
+hasCell(cellCoord:CellCoord): bool
+getCoords(): ListOfCellCoord
+getCellValue(cellCoord:CellCoord): CellValue
+getCellFormula(cellCoord:CellCoord): string
+getCellAST(cellCoord:CellCoord): ASTNode
+setCellValue(cellCoord:CellCoord,value:CellValue)
+setCellFormula(cellCoord:CellCoord,formulaStr:string)
+setCellAST(cellCoord:CellCoord,ast:ASTNode)
+removeCell(cellCoord:CellCoord)
+clone(): SSheetState
Note: state and cellData are the same datastructure
cellData contains the SSheetCell instances whereas state contains the information used to create those.
This is done because the formulas are parsed inside SSheetCell and we don't want to parse them outside of this module.
1
hasValue
1
SSheetData
-cellData: hashtable CellCoord->SSheetCell
-minCell: CellCoord
-maxCell: CellCoord
+__init__(ID:int,minCol:int=0,minRow:int=0,maxCol:int=0,maxRow:int=0)
+getLimits(): Tuple (CellCoord, CellCoord)
+getId(): int
+getState(cornerA:CellCoord,cornerB:CellCoord): SSheetState
+setState(state:SSheetState): Tuple: (errFlag, errMsg)
1
hasData
*
SSheetCell
-coords: CellCoord
-value: CellValue
-formula: string
-ast: ASTNode
+SSheetCell(cellCoord:CellCoord)
+setAST(ast:ASTNode)
+setFormula(formulaStr:string): Tuple (errFlag, errMsg)
+setValue(value:CellValue)
+getAST(): ASTNode
+getFormula(): string
+getValue(): CellValue
+getCoords(): CellCoord
getState will return a state with the cells in the range: [min(cornerA,cornerB),max(cornerA,cornerB)]
setFormula will parse the formula and create the AST. If there is a parse error, a WARNING is sent to the ErrorUtility.
<<struct>>
CellCoord
-col: int
-row: int
+CellCoord(col:int,row:int)
+col(): int
+row(): int
+hash(): int
FormulaParser

Figure 12: Preliminary interface of the `DataSubject`

–	A	B	C
1	`=B2`		
2		`=C3`	
3	`=5`		`=A3`

Suppose the formulas are evaluated in this order: `A1`, `B2`, `C3` and `A3`. To evalutate `A1`, three references must be resolved. Then, when `B2` is evaluated, the evaluator must know if `C3` and `A3` were already processed. Otherwise, it will unnecessarily resolve two references. One approach to solve this problem is to mark cells that were already evaluated. An even better approach is to evaluate the cells *topologically*. That is, to fi rst sort the cells such that they can be evaluated without having to resolve references more than once. In the above example, if `A3` is evaluated fi rst, then `C3`, `B2` and `A1`, each reference will be resolved only once. In general, such an order is obtained by sorting the cells as follows: a cell c_j is considered bigger than c_i if c_i depends on c_j. The *topological sort* approach is better because it is needed to solve another problem: cycles in the references. The evaluator will enter an infi nite loop if there is a cycle in the references. Thus, it must know in advance which cells are member of a cycle so it will not try to resolve them. Tarjan's algorithm [2] was implemented to detect cycles. The description of the algorithm can be found in Algorithm's 1, 2, 3, 4 on page 31.

The formula evaluator uses two `Visitor` subclasses, `EvalVisitor` and `InfluencerVisitor`. `InfluencerVisitor` is used just after a formula is parsed to determine the influencers of the cell (*i.e.*, the cells that are needed for evaluation). When this visitor visits a `CellRef` or a `RangeRef`, it adds the corresponding `CellCoord` to a list. A new method, `getInfluencers()` is added to `SSheetCell` interface to retrieve the CellCoord instances. With these, `EvalVisitor` is able to resolve cell references by querying `SSheetData`. Note that `EvalVisitor` assumes that the cells are evaluated in topological order. In other words, it does not call itself recursively, but retrieves the value of the referred formula, which was evaluated beforehand. The interfaces and dependencies of `EvalVisitor` and `InfluencerVisitor` are shown in Figure 13. The tasks performed by `EvalVisitor` is given in Algorithm 5. This algorithm detects errors in the formula as it tries to evaluate it. The evaluation algorithm will set the value of the erroneous cell to an error flag (to nofi ty the user without disturbing him too much). The possible errors and their respective flags are described in the following chart:

Type of Error	Explanation	Flag
Value	A function is called with non-formulas argument	`#VALUE!`
Ref	A reference is outside of the spreadsheet's boundaries	`#REF!`
Name	A function name does not exists	`#NAME!`
Cycle	The formula is member of a cycle	`#CYCLE!`

Each time `SSheetData` is modifi ed (for now, via setState), it is re-evaluated. The evaluation is done for the whole spreadsheet, though it might be possible to evaluate only the region that is affected by the modifi cation (reachability analysis). This approach will be investigated in the future. The evaluation algorithm is described in Algorithm 6.

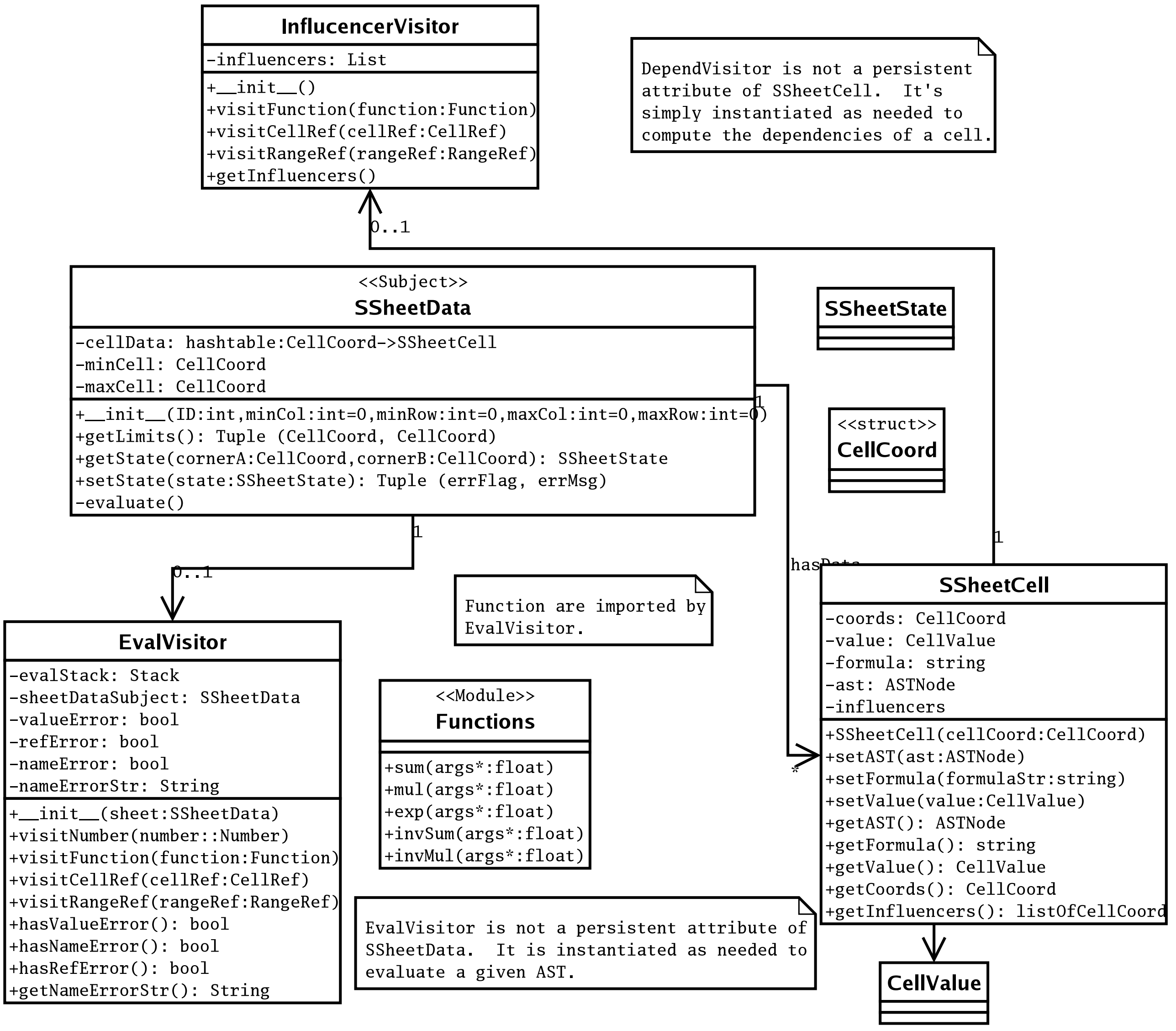

InflucencerVisitor
-influencers: List
+__init__()
+visitFunction(function:Function)
+visitCellRef(cellRef:CellRef)
+visitRangeRef(rangeRef:RangeRef)
+getInfluencers()
DependVisitor is not a persistent attribute of SSheetCell. It's simply instantiated as needed to compute the dependencies of a cell.
0..1
<<Subject>>
SSheetData
-cellData: hashtable:CellCoord->SSheetCell
-minCell: CellCoord
-maxCell: CellCoord
+__init__(ID:int,minCol:int=0,minRow:int=0,maxCol:int=0,maxRow:int=0)
+getLimits(): Tuple (CellCoord, CellCoord)
+getState(cornerA:CellCoord,cornerB:CellCoord): SSheetState
+setState(state:SSheetState): Tuple (errFlag, errMsg)
-evaluate()
SSheetState
<<struct>>
CellCoord
1
1
0..1
1
hasData
Function are imported by EvalVisitor.
EvalVisitor
-evalStack: Stack
-sheetDataSubject: SSheetData
-valueError: bool
-refError: bool
-nameError: bool
-nameErrorStr: String
+__init__(sheet:SSheetData)
+visitNumber(number::Number)
+visitFunction(function:Function)
+visitCellRef(cellRef:CellRef)
+visitRangeRef(rangeRef:RangeRef)
+hasValueError(): bool
+hasNameError(): bool
+hasRefError(): bool
+getNameErrorStr(): String
<<Module>>
Functions
+sum(args*:float)
+mul(args*:float)
+exp(args*:float)
+invSum(args*:float)
+invMul(args*:float)
SSheetCell
-coords: CellCoord
-value: CellValue
-formula: string
-ast: ASTNode
-influencers
+SSheetCell(cellCoord:CellCoord)
+setAST(ast:ASTNode)
+setFormula(formulaStr:string)
+setValue(value:CellValue)
+getAST(): ASTNode
+getFormula(): string
+getValue(): CellValue
+getCoords(): CellCoord
+getInfluencers(): listOfCellCoord
*
EvalVisitor is not a persistent attribute of SSheetData. It is instantiated as needed to evaluate a given AST.
CellValue

Figure 13: `DataSubject` with evaluate

What operations must be supported by `SSheetData` apart from modifying/retrieving the state? In the requirements, we see that at least copying and pasting a range of cells should be supported. Should `SSheetData` support such commands or should they be implemented in the user interface? We argue that the former approach is better. Suppose we want to support multiple user interfaces for different type of users. One would have to re-implement the commands for every possible user interface. It is much better to separate the data from the user interface since it improves modularity. It must also be possible to undo and redo all the operations that are applied on `SSheetData`. A naive approach to support undo/redo is to add two methods, `undoX` and `redoX`, for each command that is added to `SSheetData`. Thus, to support `paste`, the methods `undoPaste` and `redoPaste` would be added. This approach add complexity to `SSheetData` and reduce modularity. `SSheetData` is now responsible of managing the cells and the undo/redo algorithm. A better approach is to *objectify* the commands. A `Command` interface provide three methods: execute, unexecute, and reexecute. Concrete commands (such as copy, paste, etc) implement the `Command` interface and are created by a *client*. Each command has a reference to a *receiver* onto which the command is executed. In our case, the receiver is the hashtable and the client is `SSheetData`. In figure 14, we see the updated design with the Command pattern. There is a set of concrete commands, along with a special command: `CommandContainer`, which is an aggregation of commands. This class simply manages which function to call (*e.g.*, `paste.unexecute()` or `setcells.unexecute()`) depending on what command was executed last. At instantiation time, `SSheetData` creates a public `CommandContainer` object. This allows the *invoker* (the user interface) to execute commands (*e.g.*, `commands.executeCopy()`, `commands.executePaste()`). It is now possible to add new commands without modifying `SSheetData` interface.

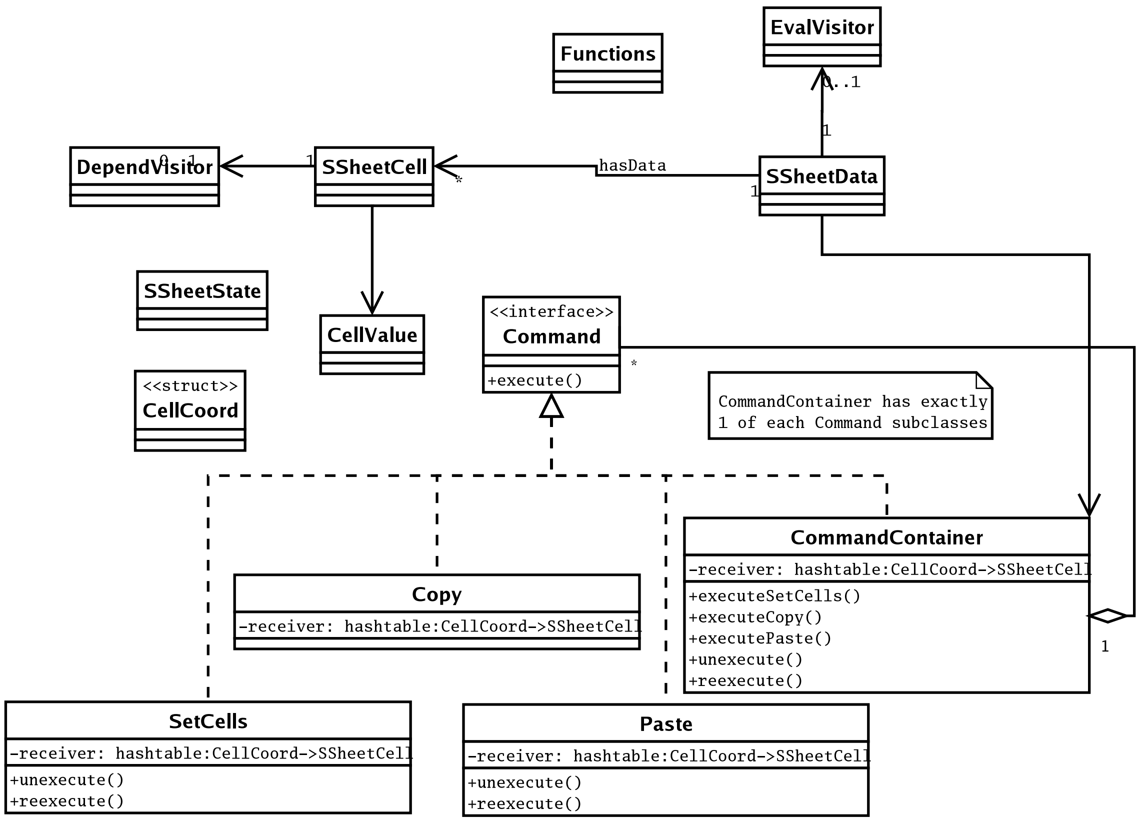


Figure 14: `SSheetData` with `Command` design pattern

The approach that was just presented goes a bit too far. First of all, at the lowest level, there is only one action that is performed on the hashtable: setting the value of a cell. For instance, the command `paste` sets the value of several cells to achieve its goal. Thus, a simpler design should contain only one command, `SetCells`, which takes a SSheetState as an argument. `SetCells` stores the cells affected by the SSheetState instance and updates the hashtable. Other commands, implemented as methods in `SSheetData`, will call `SetCells` to achieve their goal and will support undo/redo for free. This approach has three main advantages:

1. It greatly simplify the design: several command classes merged into one.
2. Avoid the need to code an undo/redo algorithm for every command.
3. It maintains modularity: all the undo/redo mechanics are still external `SSheetData`.

Now, methods must be added to `SSheetData` instead of to `CommandContainer` to support new commands. We argue that this is not less modular. The core of the algorithm that is not related to `SSheetData` (undo/redo) is implemented in `SetCells`. And all the commands that will be implemented in `SSheetData` are related to data management: move, copy, delete, set, search, replace, etc. The approach using `CommandContainer` ungroup related methods wheras the refactored approach (see Figure 15) maintains cohesion.

Now, the implementation of copy/paste needs to be discussed. References must be updated while moving a formula from one cell to another to reflect the new context. In order to achieve this, `CopyPasteVisitor` updates the cell references of the ast which is moved. It visits the ast and add the difference between the source `CellCoord` and the destination `CellCoord` to the `CellRef` instances.

`SSheetData` main role is to hold the spreadsheet data. In the next prototype, the user interface will be added and will *observe* this data. One of the requirements of DSheet is to have multiple view observing the same data. A design pattern, the *Observer Pattern*, allows an unlimited amount of *observers* kept in synch with one or more *subjects*. In our case, `SSheetData` is the subject and will

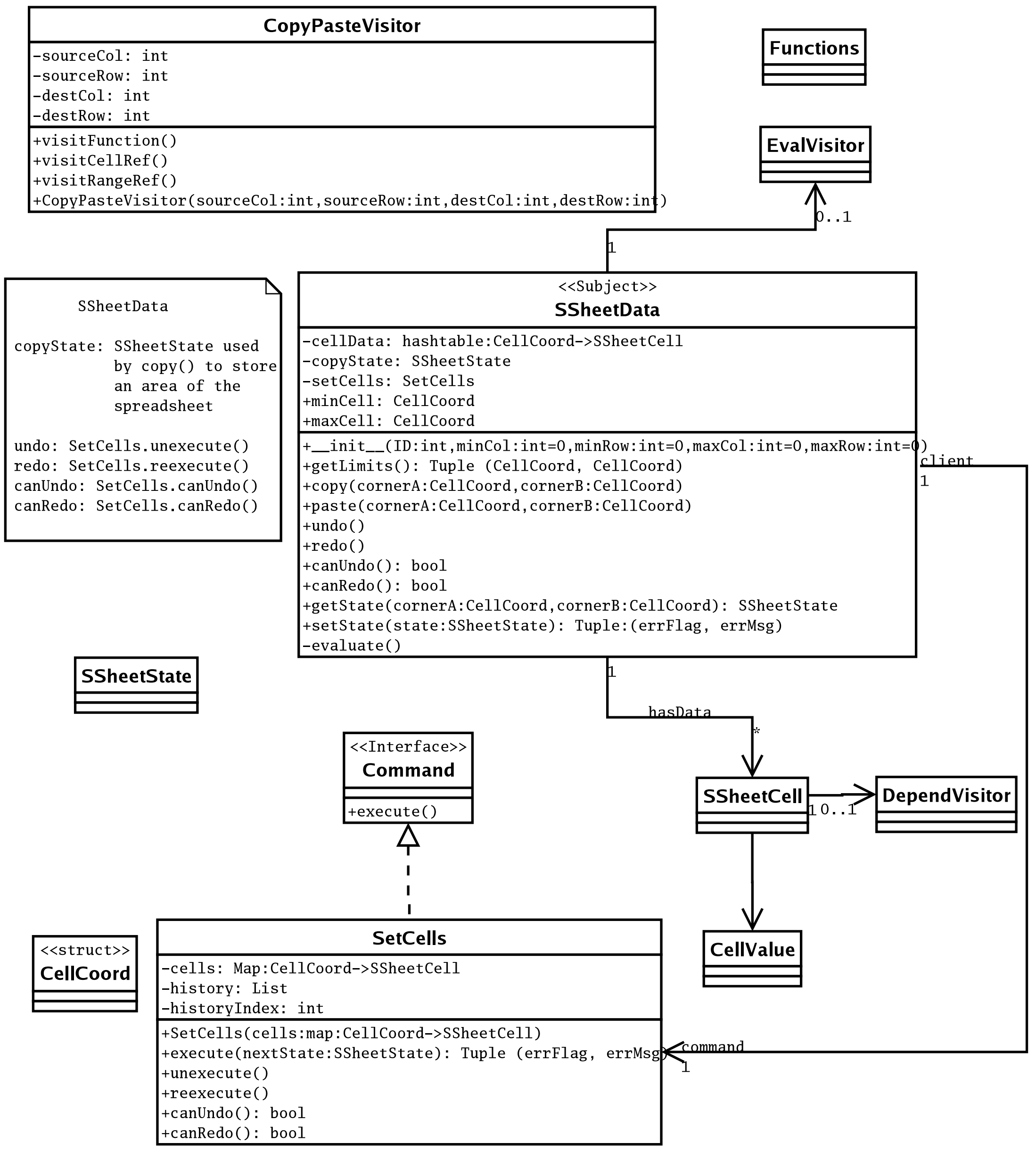

CopyPasteVisitor
-sourceCol: int
-sourceRow: int
-destCol: int
-destRow: int
+visitFunction()
+visitCellRef()
+visitRangeRef()
+CopyPasteVisitor(sourceCol:int,sourceRow:int,destCol:int,destRow:int)
Functions
EvalVisitor
0..1
1
SSheetData
copyState: SSheetState used by copy() to store an area of the spreadsheet
undo: SetCells.unexecute()
redo: SetCells.reexecute()
canUndo: SetCells.canUndo()
canRedo: SetCells.canRedo()
<<Subject>>
SSheetData
-cellData: hashtable:CellCoord->SSheetCell
-copyState: SSheetState
-setCells: SetCells
+minCell: CellCoord
+maxCell: CellCoord
+__init__(ID:int,minCol:int=0,minRow:int=0,maxCol:int=0,maxRow:int=0)
+getLimits(): Tuple (CellCoord, CellCoord)
+copy(cornerA:CellCoord,cornerB:CellCoord)
+paste(cornerA:CellCoord,cornerB:CellCoord)
+undo()
+redo()
+canUndo(): bool
+canRedo(): bool
+getState(cornerA:CellCoord,cornerB:CellCoord): SSheetState
+setState(state:SSheetState): Tuple:(errFlag, errMsg)
-evaluate()
client
1
SSheetState
1
hasData
*
<<Interface>>
Command
+execute()
SSheetCell
1
0..1
DependVisitor
CellValue
<<struct>>
CellCoord
SetCells
-cells: Map:CellCoord->SSheetCell
-history: List
-historyIndex: int
+SetCells(cells:map:CellCoord->SSheetCell)
+execute(nextState:SSheetState): Tuple (errFlag, errMsg)
+unexecute()
+reexecute()
+canUndo(): bool
+canRedo(): bool
command
1

Figure 15: `SSheetData` with refactored Command pattern

notify all of its observers each time it is modifi ed. Then, the observers will retrieve the new state of the subject and update their view. The Class Diagram of fi gure 16 shows the static structure of the observer pattern. `SSheetData` implements the `Subject` interface. Three new methods are required: `attach`, `detach` and `notify`. The two fi rst methods are used to register and unregister observers from the subject. Once an observer has registered to the subject, it will be notifi ed that the state has changed via the `nofity` method, which calls the `update` method of every registered observer.

6.4.5 Testing & Implementation

Several features need to be tested in this prototype. The approach that we have taken is to test each class independently. First, the classes are sorted by dependencies, then they are tested in that order for success, failure and sanity. The following features were tested:

- CellCoord
- CellValue
- InfluencerVisitor
- SSheetCell
- SSheetState

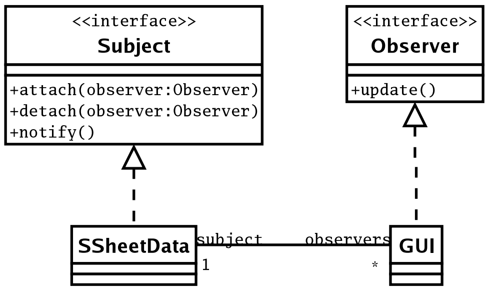


Figure 16: `SSheetData` with observer pattern

- Functions
- CopyPasteVisitor
- EvalVisitor, SetCells, SSheetData

Since `EvalVisitor` and `SSheetData` are mutually dependent, they must be tested simultaneously. Also, SetCells exposes its output only via SSheetData, so they were tested together. Each feature was tested for success by ensuring that the expected output was produced on given input. Sanity tests were applied in the last feature with undo/redo. Please refer to the fi le `/test/testsuite.py` for more details. Note that the tests are not necessary executed in the expected order (though they were written in this order) because `pyUnit` executes them in an undetermined order. Some notes about the implementation:

`CellCoord` was implemented as a class having two private attributes, `col` and `row`. Two methods, `col()` and `row()` return the corresponding values.

A `CellValue` is constructed with two facultative arguments, valueFloat and valueString. If None of them is specifi ed, the instance is undefi ned. If one of them is specifi ed, the instance is of the corresponding type, Finally, an error is raised if both of them are specifi ed (a CellValue instance is either float or string).

6.5 Prototype 4: Graphical User Interface (GUI)

6.5.1 Problem Defi nition

Now follow a description of a GUI determined with the help of the the previous use-cases and other similar tools. Since many spreadsheets are already available, the GUI of DSheet should follow their standard. It is very important, when designing a GUI, to consider the fact that the user is usually already used to a particular type of interface. In practice, it is better to stick to the usual interface patterns and not try to reinvent the wheel. If two available tools provide the same functionality, the one with the shortest learning curve will certainly be more appreciated by the user. Of course, it is still necessary to discuss these issues with the user, but by always starting with what is usually done in similar tools. In this case, the tool is far from new, so it will simply follow what is expected from such a tool. A picture (fi g. 17) from the open source spreadsheet “gnumeric” is provided to help understand the textual requirements.

6.5.2 Detailed Requirements

must-have requirements (from use-case 1):

- The interaction with the application is divided in two.
- In the top of the tool, we have the traditional pulldown menus.
- Just under the icons comes the sheet, which is a square of *m by n* cells.
- The horizontal axis is enumerated using letters (A,B,C,...)
- The vertical axis uses numbers (1,2,3,..).
- Multiple views of the same data.
- Multiple independent sheets (that could have multiple views).
- The formula of the cell under the mouse must be displayed under the pulldown menus.
- There is, at all time, a cell that is selected. Events modifying a given cell will be applied on this cell.
- Mouse left button will activate pulldown menus and icons or select a given cell
- Keyboard arrows move the selected cell in the appropriate direction. The selected cell does not move if it would exit the sheet boundaries. The Enter key move the cell down.
- Multiple cells can be selected by dragging the mouse over them.
- At least one cell will always be selected (To ensure that commands will always have a cell input). (*must-have*)

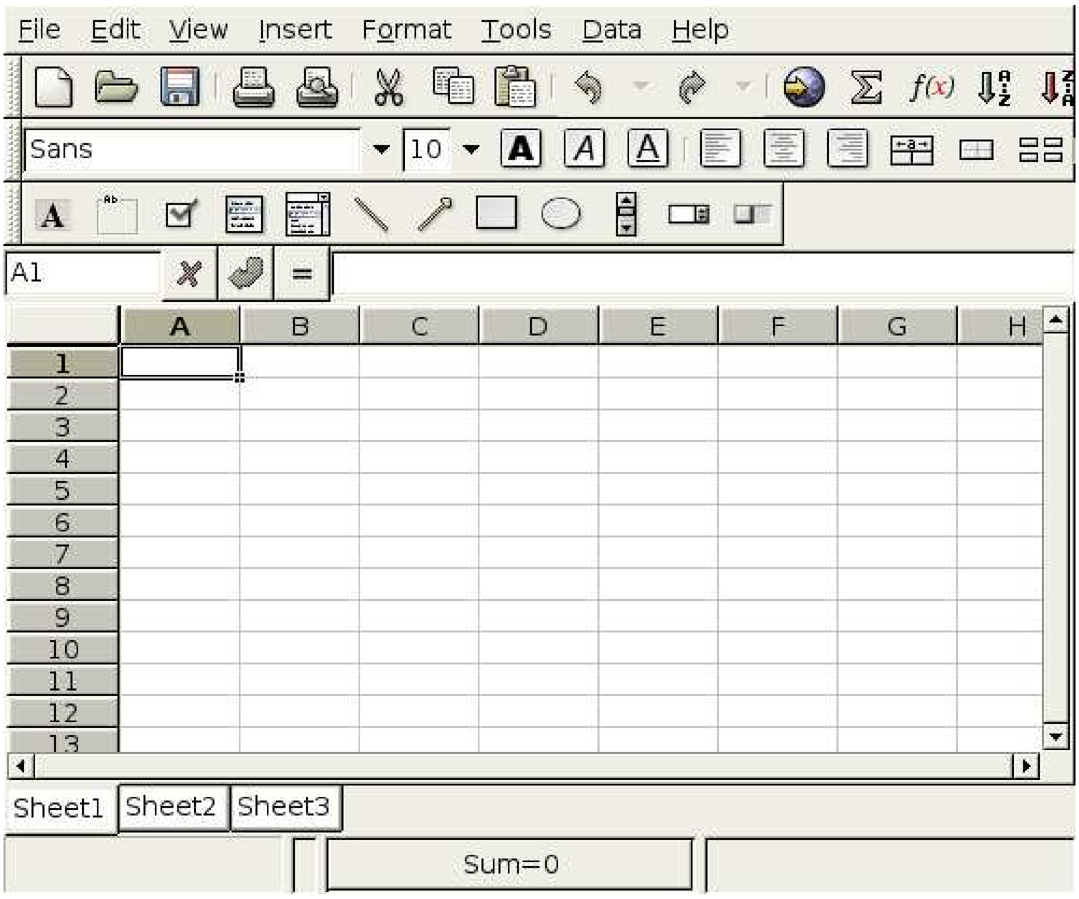


Figure 17: Gnumeric GUI

optional requirements:

- resize row/column
- format cell (font size, type, underline, etc)
- zoom
- shortcuts icons under the pulldown menus
- CTRL-F : search
- CTRL-R : search/replace

must-have requirements (from use-case 2):

- Double-clicking the left button of the mouse on a cell triggers cell editing.
- Cell editing is also activated as soon as a valid key is pressed. (A valid key is what is understood by the parser for formulas)
- There is a cursor showing where the next character will be inserted.
- The characters before and after the cursor can be deleted with backspace and delete, respectively.
- Text can be selected with the mouse, the cursor can be moved by clicking on another character.
- Any valid keyboard action delete the text selection before being applied.
- When the user is looking at the spreadsheet, the evaluated formulas are displayed. But, when a cell is edited, the actual formula is shown.

Menus Requirements:

1. File
    - New (*must-have*)
    - Open (*must-have*)

- Close (*must-have*)
- Save (*optional*)
- Save as (*optional*)
- Print (to .ps)
- Exit (*must-have*)

2. View

   - Zoom (*optional*)

3. Edit

   - Copy (*must-have*)
   - Cut (*must-have*)
   - Paste (*must-have*)
   - Undo (*must-have*)
   - Redo (*must-have*)
   - Search (*optional*)
   - Replace (*optional*)

4. Insert

   - Insert Row (*optional*)
   - Insert Column (*optional*)

Keyboard Shortcuts:

- CTRL-S : save
- CTRL-O : open
- CTRL-N : new
- CTRL-C : copy
- CTRL-X : cut
- CTRL-V : paste
- CTRL-Z : undo
- CTRL-Y : redo

6.5.3 Testing Strategy

6.5.4 Design

The graphical user interface is divided into two classes. The first class, `SSGridView_Static` implements the *static* elements of the GUI, such as menus, the spreadsheet's grid, the black rectangle indicating the selected cell, etc. This class also provides an *interface* for interacting with the static elements. This interface allows, for instance, to move the selected cell, add text to a particular cell, undo/redo operations, etc. Another class, `SSGridView_Dynamic`, calls the methods of `SSGridView_Static` when user-triggered *events*, such as mouse clicks, mouse move and keyboard keys, occur. `SSGridView_Dynamic` will be the *bridge* between the static part of the GUI and the statechart executing its behavior.

For the design of the GUI, we will assume that the implementation is using a library supporting the following features:

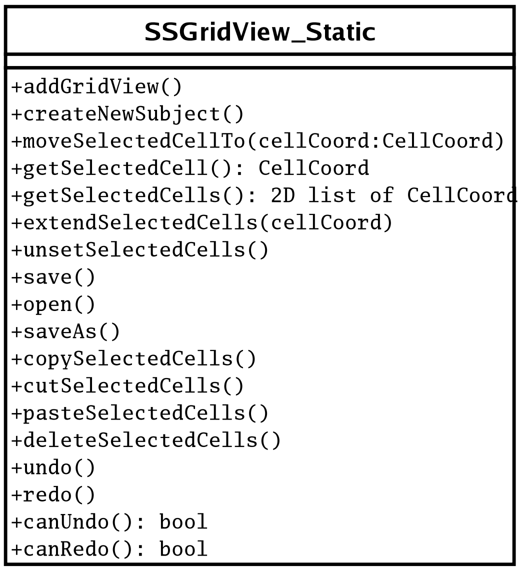
- A canvas, where it is possible to draw geometric shapes such as lines, rectangles and circles. It must also be possible to draw text.
- Built-in pull-down menus.
- System to handle keyboard and mouse events. More specifically, we will assume that, when it occurs, an event is passed to a programmer-specified method call that will handle it.

The interface of `SSGridView_Static` is directly determined by the requirements specified above. At the highest level, the requirements can be splitted into two categories: *viewing* the spreadsheet and *editing* the spreadsheet. While the user is viewing the spreadsheet, the following actions can be performed:

- Move the selected cell

- Select cell(s) and apply commands on them:
  - Copy
  - Cut
  - Paste
  - Delete
- Activate pull-down menus:
  - New (new subject)
  - Add Gridview (add observer)
  - Print to .ps
  - Copy
  - Cut
  - Paste
  - Undo
  - Redo
- Start editing a cell

Thus, the interface of `SSGridView_Static` must have methods implementing those actions, as seen in the Class Diagram of figure 18.



```
SSGridView_Static

+addGridView()
+createNewSubject()
+moveSelectedCellTo(cellCoord:CellCoord)
+getSelectedCell(): CellCoord
+getSelectedCells(): 2D list of CellCoor
+extendSelectedCells(cellCoord)
+unsetSelectedCells()
+save()
+open()
+saveAs()
+copySelectedCells()
+cutSelectedCells()
+pasteSelectedCells()
+deleteSelectedCells()
+undo()
+redo()
+canUndo(): bool
+canRedo(): bool
```

Figure 18: Class Diagram including the methods used while *viewing* the spreadsheet

A few notes about the methods:

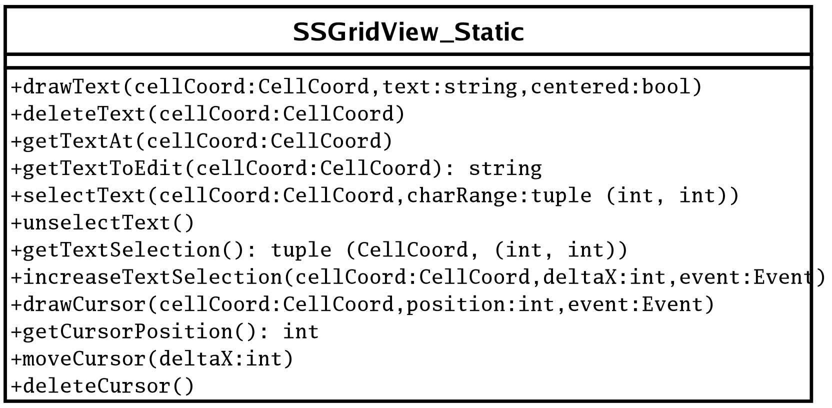
- `addGridView` could be used in conjunction with another method to query the user for parameters, such as width, height, cell width, cell height, etc. The same technique could be applied with `createNewSubject`
- If the parameter passed to `moveSelectedCellTo` is outside of the bounds of the subject, an error is raised.
- `extendSelectedCells` increase the range of selected cells to include `cellCoord`, with respect to a *pivot*. The pivot is a cell coordinate that is always a boundary of the range of selected cells. There is always a trivial range of cells that is selected: the selected cell. When `extendSelectedCells` is called for the first time, a range of cell (selectedCell, cellCoord) is created. The pivot is set to selectedCell. Commands can be applied on this range, or it can be *resized*. When the range is resized, the pivot is used to create a new range of cells that is coherent with the previous one. `unsetSelectedCells` simply deselects the cells and resets the pivot.
- `pasteSelectedCells` pastes the previously copied cells into the current selection.
- `undo`, `redo`, `canUndo`, `canRedo` are directly bound to the corresponding methods of `SSheetData`.

While the user is editing the spreadsheet, he is really editing a particular cell. The following actions can be performed when the user edit a cell:

- Add a new character

- Remove the previous or the next character
- Select character(s) and apply commands on them:
  - Delete
- Commit the data and start viewing the spreadsheet

Again, the interface of `SSGridView_Static` must support these actions, as seen in the Class Diagram of figure 19.



```
SSGridView_Static

+drawText(cellCoord:CellCoord,text:string,centered:bool)
+deleteText(cellCoord:CellCoord)
+getTextAt(cellCoord:CellCoord)
+getTextToEdit(cellCoord:CellCoord): string
+selectText(cellCoord:CellCoord,charRange:tuple (int, int))
+unselectText()
+getTextSelection(): tuple (CellCoord, (int, int))
+increaseTextSelection(cellCoord:CellCoord,deltaX:int,event:Event)
+drawCursor(cellCoord:CellCoord,position:int,event:Event)
+getCursorPosition(): int
+moveCursor(deltaX:int)
+deleteCursor()
```

Figure 19: Class Diagram including the methods used while *editing* the spreadsheet

A few notes about the methods:

- `getTextToEdit` returns the text that needs to be edited at `cellCoord`. It queries the subject for the formula string or the string value that is located at `cellCoord`.
- Text selection is specified in range of *characters*. Note that if multiple fonts are used, calculations need to be performed to draw the range correctly.
- In `drawCursor` and `increaseTextSelection`, `event` overrides `position` and `deltaX` respectively when specified. For instance, it is possible to draw the cursor where a mouse click event occured.

There is also a set of aesthetic requirements that need to be fulfilled. The cell which is under the mouse will be indicated to the user by drawing a light blue rectangle around the cell. We will call this entity the *over cell rectangle*. The coordinates of the cell under the mouse will be displayed in a label on the top of the sheet. Similarly, the formula or the string value of the cell under the mouse must be displayed in a label on the top of the sheet. Four methods, `moveOverCellTo`, `getOverCellCoord`, `setOverCellTextDisplay` and `setOverCellCoordDisplay` must be added to fulfill these requirements. Note that the overcell rectangle is always drawn around the cell closest to the mouse pointer, even if the pointer is outside of the sheet.

The next step in the design is to model the behavior of the graphical user interface. As mentionned previously, the GUI has two high level states: *viewing* and *editing*. Depending in which state the GUI is, a set of keyboard and mouse events can occur and a subset of this set will affect the application. For instance, we know from the requirements that a double left click on a cell triggers editing whereas a double right click does nothing. Thus, we must determine the events that are used in *viewing* and *editing* and model how they affect the GUI. Next follows a list of meaningful events while *viewing* the GUI:

- Mouse Events:
  - `mouseClickL`: single left button click
  - `doubleMouseClickL`: double left button click
  - `mouseMove`: mouse move
  - `mouseReleaseL`: left button release
- Keyboard events:
  - `editKey`: =, a-z, A-Z, 0-9, / !@#$%^&*()-_+;:’”,<>?[]
  - `upKey`, `downKey`, `leftKey`, `rightKey`: corresponing keyboard arrows
  - `shiftPress`: right or left shift press
  - `shiftRelease`: right or left shift release
  - `controlC`, `controlV`, `controlX`, `controlZ`, `controlY`: holding control button and pressing the corresponding key.

The statechart of Figure 20 models the behavior of a user viewing the spreadsheet. Three states were needed, `NotSelecting`, `SelectingKeyboard` and `SelectingMouse`. Each time a transition is fi red in the statechart, a particular action must be executed. For instance, if the event `mouseMove` occurs while the statechart is in `Viewing.NotSelecting`, the overcell rectangle must be updated to reflect the new context. Similarly, if the event `mouseClickL` occurs in the same state, the selected cell must be moved to the cell closest to the event. The interface of `SSGridView_Static` will be used to implement these actions. Two events, `editKeys` and `doubleMouseClickL` trigger the editing mode, which is modelled in Figure 21.

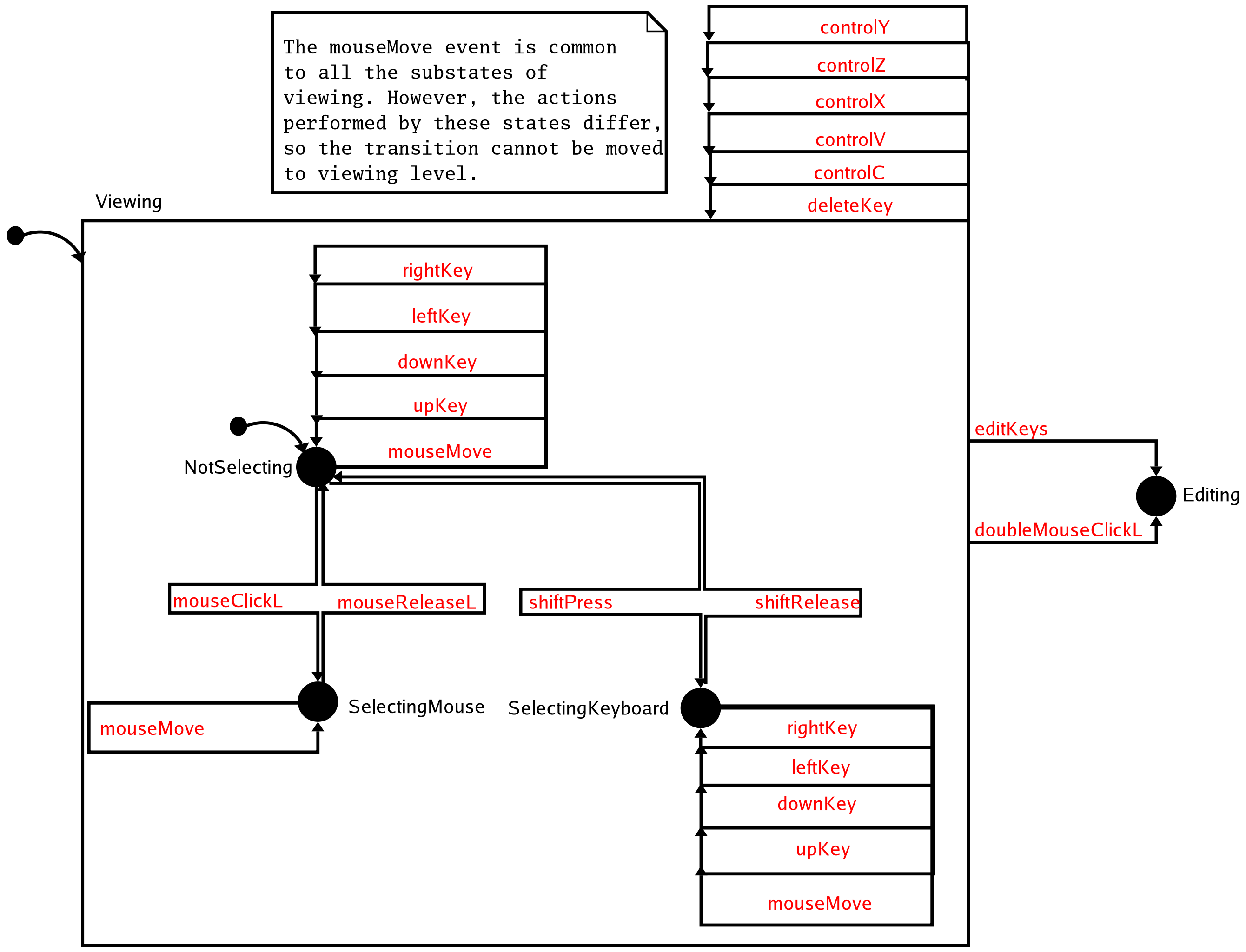


Figure 20: Statechart modelling the behavior of a user viewing the spreadsheet

In the following chart, the actions, which are executed after an event occurs, are described. A few notes about the chart. First, there is no guard that ensures that there is a copy buffer when copying the cells, because copying an empty buffer has a meaning: copy nothing. Also, canRedo() and canUndo() are used as guard to be able to enable/disable the shortcuts in the menus, as in professional tools.

- Viewing
  - Event controlY
    - Guard: canRedo()
    - Action:
      1. Redo the next action.
      2. If cannot redo anymore,disable the corresponding button in the menu.
      3. Enable undo button
  - Event controlZ
    - Guard: canUndo()
    - Action:

    1. Undo the previous action.
    2. If cannot undo anymore, disable the corresponding button in the menu.
    3. Enable redo button

- Event controlC
  - Guard: None
  - Action:
    1. Copy the selected cells
- Event controlX
  - Guard: None
  - Action:
    1. Copy the selected cells
    2. Delete the selected cells.
    3. Enable the undo button
- Event controlV
  - Guard: None
  - Action:
    1. Paste the copy buffer into the selected cells.
    2. Enable the undo button
- Event deleteKey
  - Guard: None
  - Action:
    1. Delete the selected cells
    2. Enable the undo button
- Event editKey
  - Guard: None
  - Action:
    1. Draw the pressed key in the selected cell.
    2. Draw the cursor in the selected cell.
- Event doubleMouseClickL
  - Guard: None
  - Action:
    1. Move the selectedCell to where the click occured
    2. Draw the cursor
- Event leftKey
  - Guard: `selectedCell.col()` $>$ `0`
  - Action:
    1. Move the selected cell rectangle

6.5.5 Implementation & Testing

The GUI was implemented using python's built-in `Tkinter` module. The structure of the implementation follows closely the design presented in the previous section. The details of the implementation will be discussed here, in hope that it is useful for other implementations. The following actions are performed when `SSGridView_static` is instantiated. First, high level widgets are packed together: the spreadsheet canvas where the cells reside, scrolls to move the canvas, menus and fi elds used to display the formula and the current cell. Then, a set of private methods contruct the inital aspect of the GUI. Vertical and horizontal lines are drawn on the canvas to create the set of cells [1]. Two private array of integers, `rows` and `columns`, keep track of the position of every line, to support resizing in the future. The private methods `drawRows` and `drawColumns` use those arrays to display the lines. Two private variable hold the coordinates of the selected and over cell. Two corresponding private methods, `drawOverCell` and `drawSelectedCell`, can be called to update the display by (re)drawing the rectangles at those coordinates. Other privates function are used to draw the column and row indexes (*i.e.,* A,B,C, 1,2,3) and their background. To manage text on the canvas, a hashtable maps `CellCoord` instances to `Text` objects, which are built-in in `Tkinter`. There is also a similar hashtable for `Font` objects, anticipating customizable fonts in the future (right now, the same font is used for every cell).

[1] The very fi rst prototype used a rectangle. for every cell of the spreadsheet, but this approach turned out to be very ineffi cient.

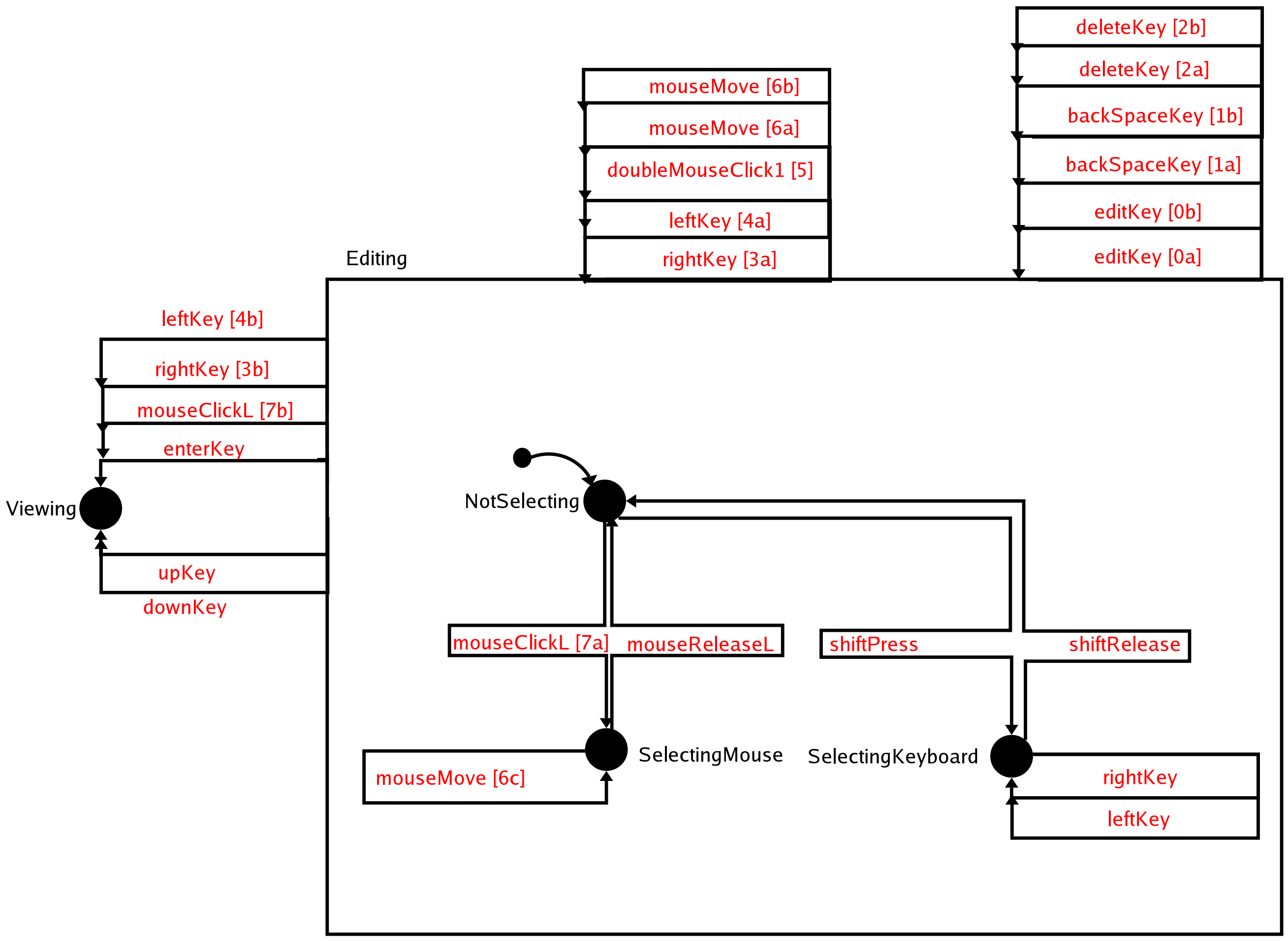


Figure 21: Statechart modelling the behavior of a user editing the spreadsheet

`SSGridView_static` also hold a reference to a `SSGridView_dynamic` instance. This instance is the *bridge* between the static part of the GUI and the DChart executing its behavior. `Tkinter` events are binded to the methods of `SSGridView_dynamic`, which in turn send the event to the DChart executed by SVM [2] [1].

7 Conclusion & Future extensions

Currently, the requirements specify the need for multiple views of the same sheet, but only locally. It would be very nice to have a client/server mechanism supporting multiple network clients viewing the same data. This would involve some synchronization, but it should be easy to extend the current local approach to a networked approach. Other observers, such as a plotter, will probably be needed in a future version. The GUI should also support the following features for it to be professional:

- Resize columns and rows
- Add new columns or rows in-between existing ones
- Change layout properties of cell's border
- Print
- Cell text format
- Zoom

It would also be very nice to have statistical analysis of the spreadsheet data and a linear programming solver.

[2]http://msdl.cs.mcgill.ca/people/tfeng/?research=svm

References

[1] Huining Feng. Dcharts, a formalism for modelling and simulation based design of reactive software systems. Master's thesis, Mcgill University, February 2004.

[2] R.E. TARJAN. Depth fi rst search and linear graph algorithms. *SIAM J. Comptg.*, 1:146–160, 1972.

Algorithm 1 topoSort

```
global cells {datastruct containing the cells}
local mapping cellsLabelling
local list sortedList
for each cell in cells do
    cellsLabelling.put(cell, NOT_VISITED)
end for
for each cell in cells do
    if cellsLabelling.get(cell) = NOT_VISITED then
        dfsSort(cell, cellsLabelling, sortedList)
    end if
end for
return sortedList
```

Algorithm 2 dfsSort(cell, cellsLabelling, sortedList)

```
if cellsLabelling.has(cell) and cellsLabelling.get(cell)==NOT_VISITED then
    cellsLabelling.put(cell, VISITED)
    for influencer in getInfluencersOf(cell) do
        dfsSort(influencer, cellsLabelling, sortedList)
    end for
    sortedList.append(cell)
end if
```

Algorithm 3 strongComponents

```
global cells
sortedList = topoSort()
local mapping cellsLabelling
local list strongComponents
for each cell in cells do
    cellsLabelling.put(cell, NOT_VISITED)
end for
for each cell in sortedList.reverse() do
    if cellsLabelling.get(cell) = NOT_VISITED then
        local list component
        dfsCollect(cell, cellsLabelling, component)
        strongComponents.append(component)
    end if
end for
strongComponents.reverse() {return strong components in topological order}
return strongComponents
```

Algorithm 4 dfsCollect(cell, cellsLabelling, component)

```
if cellsLabelling.has(cell) and cellsLabelling.get(cell)==NOT_VISITED then
    cellsLabelling.put(cell, VISITED)
    for dependent in getDependentsOf(cell) do
        dfsCollect(dependent, cellsLabelling, component)
    end for
    component.append(cell)
end if
```

Algorithm 5 EvalVisitor

```
local evalStack
if astNode is Number then
    Push the float value on evalStack
end if
if astNode is CellRef then
    if the CellRef is valid then
        Resolve the cell reference.
        if the cell does not exists then
            Push the value 0.0 on evalStack
        else
            Otherwise, push the float or string value on evalStack
        end if
    else
        Set reference error to true
        Push the value 0.0 on evalStack
    end if
end if
if astNode is RangeRef then
    Recusively visit each cellref in the range.
end if
if astNode is Function then
    Recursively visit each argument of the function
    Pop the appropriate number of arguments from evalStack
    if all the arguments are float values then
        if the function exists then
            Apply the function on the arguments
        else
            Set Name error to true
        end if
    else
        Set value error to true
    end if
end if
```

Algorithm 6 Evaluate

```
global SSheetData sheet
sortedComponents = strongComponents()
local cycles
for component in sortedComponents do
    if len(component)==1 then
        local cell = component[0]
        if cell in getInfluencers(cell) then
            cell.setValue(0.0)
            cycles.append(component)
        else
            if cell.getAST()!=None then
                v = EvalVisitor(sheet)
                cell.getAST().accept(v)
                if v.hasError() then
                    if v.hasValueError() then
                        cell.setValue(VALUE_ERROR_FLAG)
                    end if
                    if v.hasNameError() then
                        cell.setValue(NAME_ERROR_FLAG)
                    end if
                    if v.hasRefError() then
                        cell.setValue(REF_ERROR_FLAG)
                    end if
                else
                    cell.setValue(v.getEvaluatedValue())
                end if
            end if
        end if
    else
        cycles.append(component)
        for cell in component do
            cell.setValue(0.0)
        end for
    end if
end for
for cycle in cycles do
    for cell in cycle do
        cell.setValue(CYCLE_ERROR_FLAG)
    end for
end for
```