# Genealogic tree: Report

## Table of content:

# I/ Genealogic tree: Summury

The aim of the projet was to create  a console program allowing a user to interact with genealogic trees. This was a single person project. You will find in this report a description of the program at all the levels of abstraction, from the specifications down to the actual implementation and the tests. I will also tackle the dificulties I faced while developing it, what I learned from them and I will give you a brief idea of what could be upgrade or added to the program.

# II/ Strategy

## A- Tools

I developped this program in *Ada*. I used *visual studio code* as my integrated development tool and gnat as a compiler. All the project, including the code, tests and documentation were kept in my personal *github* account.

## B- Development cycle

Before going down to the actual coding, I dedicated an hour to the specification reading, putting in place the global folder structure of the project that was provided, and thinking about the different data types and their interactions.
This project was really similar to others I did at the university, so I quickly had a good idea of the project as a whole.

I divided the whole project into smaller goals achievable in a typical 2 hours session time. I added progressively all the features of the program when I was certain that the existing ones worked properly. This allowed me to have a program always ready to run, but made me re-organized files/functions a couple of time to avoid redundancy.

Concerning tests, I did  them at the end once everything was coded. While developping features, I wrote a list of possible failures so that I can test them later on. I should point out that writing tests  let me spot errors I did not see.

## C- Principles

While designing the program, I kept in mind:

- The single responsibility principle, so that each file/function has it own purpose. The aim is to avoid large files and allow to easily add features to the program (See the arctitecture section). For the functions, this resulting in the *raffinage* process learned it school.

- code encapsulation. I made sure subtypes were private when necessary, and gave functions/procedures to interact in a defined and controlled way.

- genericity, so that part of my code could be reused for futur project.

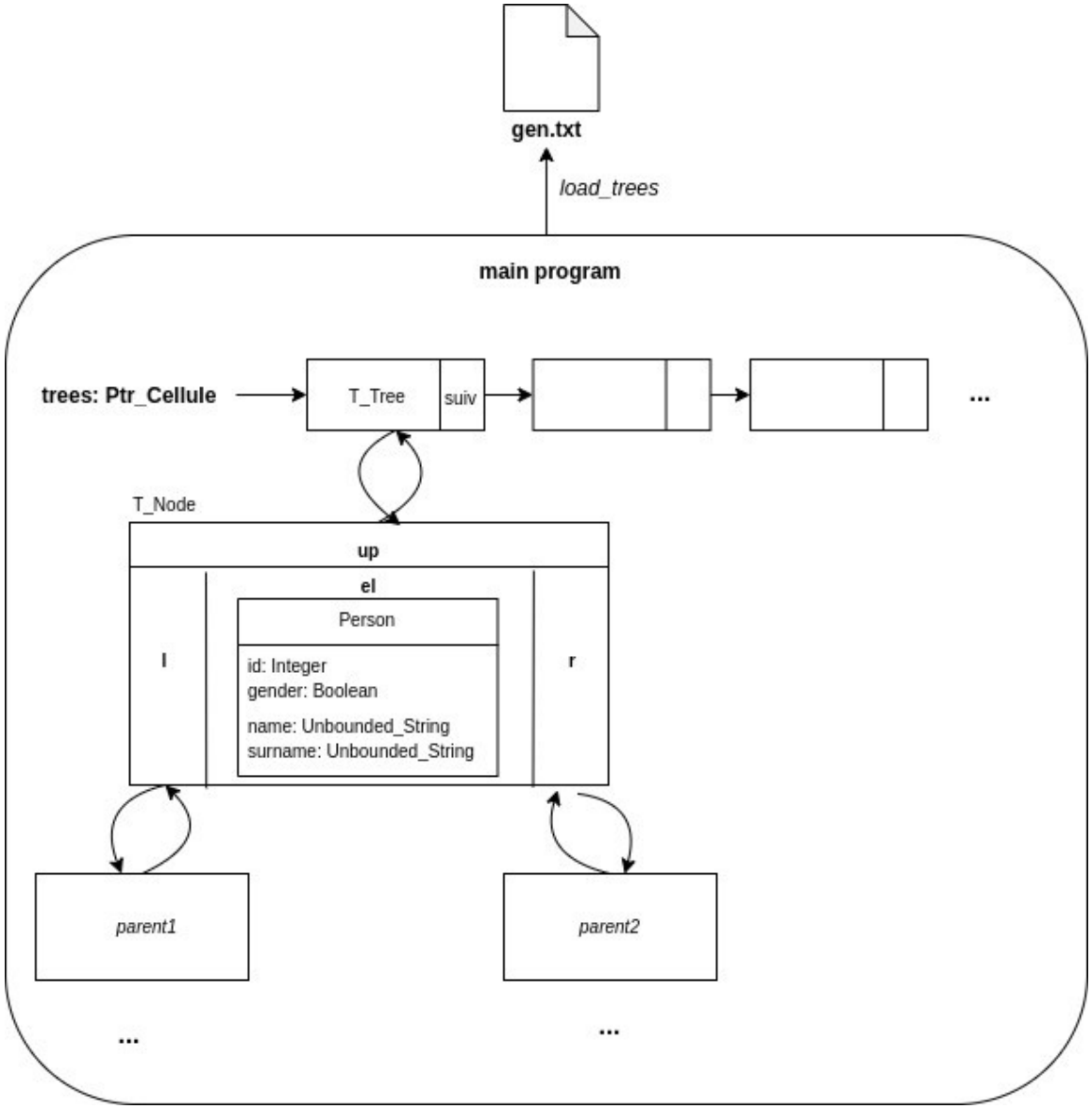
## III/ Architecture


*Genealogic tree module architecture*

Warning:
*gen* refers to the genealogic term in all files except p_list_gen. Here *gen* means *generic*.

This graph is pretty straightforward, it should be read bottom to top. The *start* file launch *main*. *Main* presents the user with all the available options by using the *console* module. *Main* manipulates concrete data-types that derived from generic ones. *utils_gen* provides other functions to manipulates list of trees, to save and load them for instance. The *utils* and *alea* are not specific to this project and were used as a helper throught the project.

## IV/ Data-Types




*Genealogic tree data organisation*

The data is organised as such.
- The *main.adb* contains one *trees* variable, which is a chained list of *T_Tree*.

- Each tree represents the genealogic tree of a person. It is a binary tree data structure pointing to a node *T_Node*.

- *T_Node* is a structure containing the actual person, one pointer for each parent (*l* and *r*) and a pointer to it descendant (*up*). I choosed to introduced *up* due to the specifications. Indeed, some features of the program were about showing descendants of someone. Having this pointer let the algorithms being simpler.

- *Person* is a structure containing the person's info.

- The variable *trees* can be loaded from a formatted text file via the *load_trees* methods (more on that in the algorithm section).

## V/ Algorithmes

In this section, I will explain some functions that desearved a detailed explanation. For the other ones, please refer to the comments directly in code.

| File | Porpuse | Steb by step explanation |
|---|---|---|
| `procedure enlever(e: in T_Element; l: in out Ptr_Cellule) with Post => not exist(e, l);` | | |
| p_list_gen | Remove element e from list l. | Iterative.<br>- Check if it is the first element to remove. If so, it just returns the next element.<br><br>-Otherwise, check out the next element. If the next element if the one to delete, change it to the one after.<br>Otherwise, loop until you find it or you reach the end. |
| `function stringify_person(el: in T_Person; ancestor_id: in Integer := -1) return Unbounded_String;` | | |
| person | Convert a person into a string (to save him into a file later). | - Some person doesn't have descendant (root of the tree). Let stringify know by providing a -1 for ancestor_id.<br><br>Create an Unbounded_String str.<br><br>- concat the ancestor id to str if it is not -1.<br><br>- concat the gender to str (0 for man, 1 for woman)<br><br>- concat the name and surname to str<br><br>- return the Unbounded_String. |
| `function person_from_line(line: in Unbounded_String; ancestor_id: in out Integer; root: in Boolean) return T_Person;` | | |
| person | Convert a string into a person (the reversed process of stringify_person). | Iterative.<br>- The string *line* is read character by character. To isolate a field of the string, we look for a white space. When found, the corresponding field is the substring starting from *last_index* to *current_index*.<br><br>- El: Integer is the index of the reading part of the string. It helps to know which field of the person we are reading.<br><br>- So we loop throught each character of *line*. Isolate the field when there is a whitespace. See what is that field via a switch on *el*. And save the info into a person via some conversion.<br><br>- We return the final person. |
| `procedure show_descendant(tree: in T_Tree; descendant: in Integer; cur_desc: in Integer := 0);` | | |
| tree_bin | Show all descendant at a given level. | Recursive.<br>- We stop when a tree is empty. |

| | | |
|---|---|---|
| | | - If this is the right level or if level is -1, we show the person info for the current tree.<br><br>- We loop iteratively to the descendant by using the *up* pointer, incrementing the level. |
| function stringify_tree(tree: in T_Tree; ancestor_id: in Integer := -1) return Unbounded_String; | | |
| tree_bin | Convert a tree into a tree. | Recursive.<br>- If the tree is empty, return an empty string.<br><br>- Otherwise, return the stringify person of that tree, followed by the stringify version of the 2 subtrees. |
| function find_el_trees(id: in Integer; trees: in Ptr_Cellule; root: in Boolean := false) return T_Tree; | | |
| utils_gen | Find a tree element inside a chained list of trees. | Iterative.<br><br>- For each tree of the trees: Ptr_Cellule,<br><br>- try to find the element with id inside that tree.<br><br>- If found, we can whether return the subtree containing the element at id, or we can return the whole tree (based on the use of the function).<br><br>- Otherwise, find the element in the next tree of the list. |
| function save_trees(trees: in Ptr_Cellule; file_name: in Unbounded_String) return Boolean; | | |
| | | |
| function load_trees(file_name: in Unbounded_String) return Ptr_Cellule; | | |
| | | |
| procedure user_program(trees: in out Ptr_Cellule); | | |
| | | |

## VI/ Tests

## VII/ Difficulties

## VIII/ What’s next ?

Real id
new information kept into person

graphical interface

## IX/ Personal sumup