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.

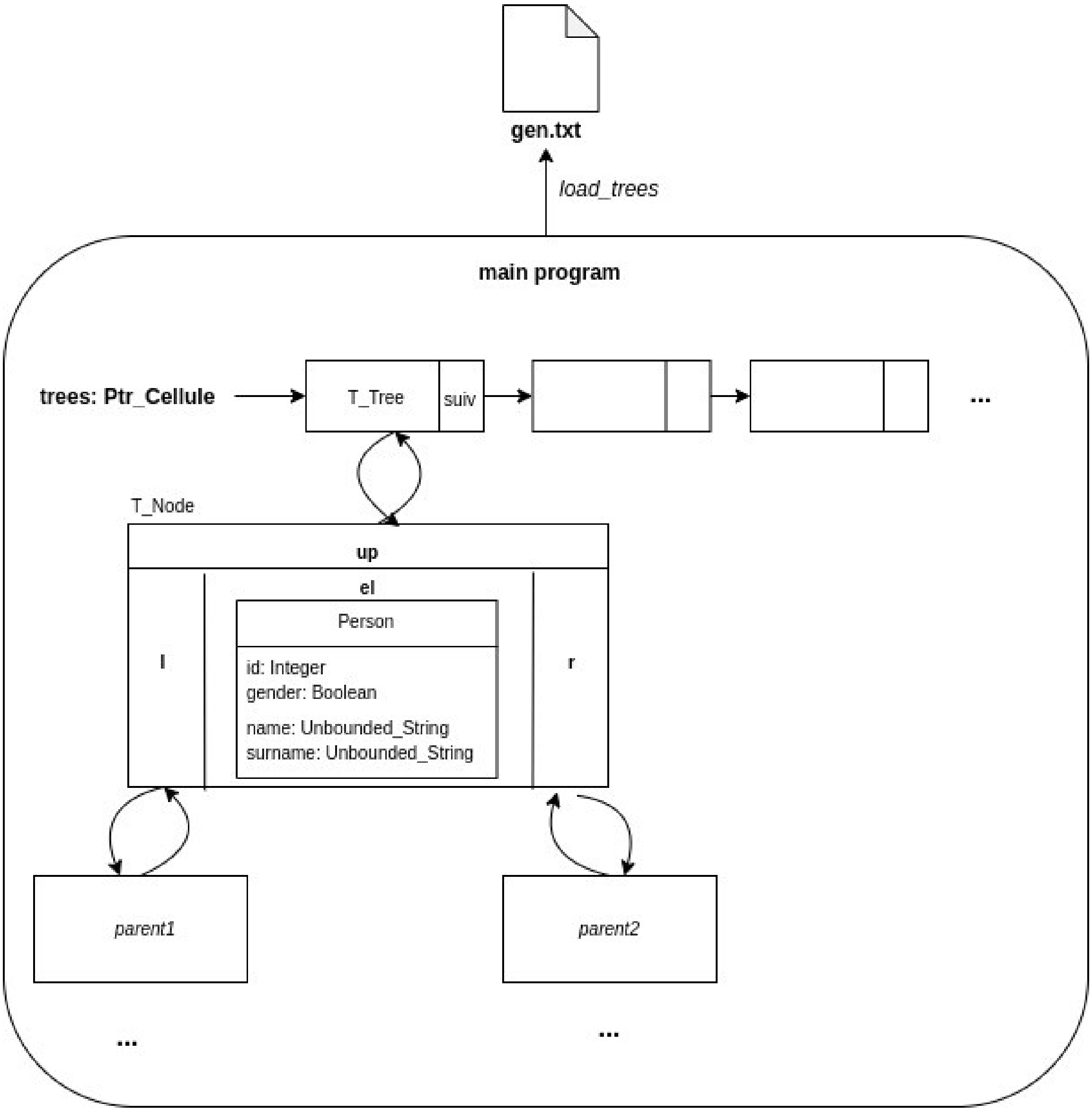
III/ Architecture

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 to write and with a lower complexity.

- *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 /src

In this section, I will explain some functions that desearve a detailed explanation. For the other ones, please refer to the comments directly in code. I used both iterative and recursive functions/ procedures throuhout the project, depending mostly on my subjective code style.

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. - Check if it is the first element to remove. If so, it just returns the next element. -Otherwise, check out the next element. If the next element if the one to delete, change it to the one after. 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. Create an Unbounded_String str. - concat the ancestor id to str if it is not -1. - concat the gender to str (0 for man, 1 for woman) - concat the name and surname to str - 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. - 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*. - El: Integer is the index of the reading part of the string. It helps to know which field of the person we are reading. - 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. - 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. - 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. - 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. - If the tree is empty, return an empty string. - 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. - For each tree of the trees: Ptr_Cellule, - try to find the element with id inside that tree. - 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). - 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;`		
utils_gen	Save a chained list of trees inside a text file.	Iterative. Create an empty Unbounded_String *trees_str*. - Go throught each tree, and concat *trees_str* with the *stringify_tree* procedure. - Open/create a text file with the provided file_name - Write *trees_str* into that file - Close the file.
`function load_trees(file_name: in Unbounded_String) return Ptr_Cellule;`		
utils_gen	Create a chained list of trees Ptr_Cellule out of a text file.	Iterative. -Create an empty Ptr_Cellule. - Open the file and loop over each line. -If the line is “arbre”, that mean that the following lines correspond to a new tree. So we get the root person out of the next line, create a new tree with him and insert that tree into the list of trees. - Otherwise, we create a new person out of the line and get his ancestor id. This id is used to insert this person at the right place.

<table>
<tr><td></td><td></td><td>- We close the file and return the tree.</td></tr>
<tr><td colspan="3"><code>procedure user_program(trees: in out Ptr_Cellule);</code></td></tr>
<tr><td>main</td><td>Program interacting with the user to let him manipulates genealogic trees.</td><td>The trees parameters is the variable holding all the data (chained list of trees). It is a parameter for testing purposes (more on that in the tests section).

This program is an infinite loop. One iteration corresponds to one user action in the system. For each iteration:

- We show every option the user has and get his choice, using the console module.

- Do the right action via a switch. Each action consist of getting more user input if necessary and call the diferent module’s procedures/functions on our trees variable.</td></tr>
</table>

VI/ Tests

In /livrables/tests

Each module has it own test module, For instance, utils.ads → test_utils.ads. A test module consists of one test per function/procedure. The point is to make sure every function work as expected by delivering the correct result.
One test consist of:
- A prepare phase: Initialising variable we need as parameters.
- Running phase: Calling the function to test with those variables.
- Assert phase: Comparing the result given by the function to what is expected, using the *assert* procedure.

A test passes if the assert is correct, it fails otherwise.

tests.adb is the starting point. It includes every test modules, call them and is successful if every test passed. The order it calls the tests matter. Indeed, we first test the most generic functions available, aka the ones that don't depend on the others. By doing this, we ensure that a bug in a function come from this function and not subfunctions it uses. That why the test of *utils* come first. Then, the order follows the *Genealogic tree module architecture* graph, going from top to bottom for the exact same reason.

let's have a look on every test files:

tests.adb

Name	**Strategy**
tests_utils	Nothing particular. It is a typical test file on it simplest form.
tests_tree_bin	Here, tree bin are tested containing Integer and not Person. This does not matter as what we test here is the behavior of the binary tree <u>generically</u> speaking. Furthermore, using simple Integers allowed to greately simplify our tests. Again, each procedure/function is tested following the specification order.
tests_person	Here, *stringify_person* and *person_from_file* are supposed to have the inverted behavior. We use that to test both functions at once. Don't worry, in case of a bug we are still able to tell which function has a problem.
tests_console	Here, the tester is asked to enter specific user input supposed to cover all the diferent possibility. The tester is the one judging if a test Is successful or not. This could leads to some problems discussed in the *What's next* section.
tests_utils_gen	Here, *save_trees* and *load_trees* are also complementary. The tester is also the one judging the success for some tests.
tests_main	All the other tests above could be considered as unit tests, as they focus on one function/procedure. The test of main was thought as an integration test. The tester is placed in the user's shoes and is guided to follow a defined scenario supposed to test every features of the program. Basically, the tester is asked to enter defined user input, and is telled what should be the resulting consequences. By passing the *trees* as an *in out* parameter in the *user_program*, I was able to do most of the assert based on that variable. However, the tester help was still required.

VII/ Difficulties

No major difficulties. As I said, I was familiar with those kind of program and everything flew as expected.

The difficulties mostly came from ADA due to it differences with more common programming languages like C or java. For instance:

How the use a concrete type derived from a generic module in different file. Essential to divide the code into multiple files. How to deal with strings to cut them into substring.

The persistence part was the trickiest. Doing it was optional and made me add new functions that changes (for the good) the way the project was organised.

How tests should be organised.

How to use defined user input during tests.

As a side note, we have to include the *src/* folder containing the modules to test. This is done by adding the *-I* parameters in the compiler commands (more on that in the manuel section). The resulting command to compile is:
$ *gnatmake* -gnatawa -I../../src -gnata -g tests.adb

VIII/ What’s next ?

Real id
new information kept into person

graphical interface

fully automised test that doesn’t required any human hands.

IX/ Personal sumup