Design Document for “Visual Family Tree”

Kaya Oğuz

1 Introduction

Visual Family Tree is a standalone desktop application that can create, edit, display and merge family trees. The application is targeted for Windows desktops. The application will be in Turkish. Throughout the document it is assumed that Java programming language will be used for implementation.

These meet the following non-functional requirements.

Non-functional Requirement 2: The system shall be able to run on a Windows system.
Non-functional Requirement 3: The system shall be in Turkish.

2 Software Architecture

Since it is a standalone desktop application that does not rely on any other application or service, Visual Family Tree is planned to have a basic software architecture: an internal set of classes will handle the family tree operations, and a visual representation of the tree will be provided for the graphical user interface. In this regard, it would be best to begin with the classes that make up the family tree.

2.1 Structural Design

Any family tree relies on the fundamental relationship between the parents and the children. All relatives are defined according to the parents and their own siblings. So, two classes, Person and Relation, are designed to handle this relationship.

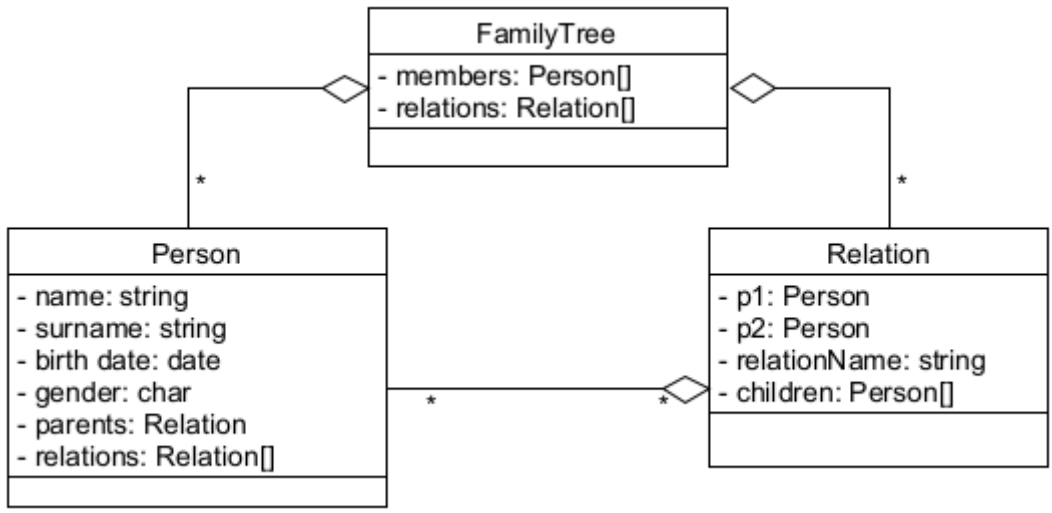


Figure 1: Class diagram for FamilyTree, Person and Relation classes.

As can be seen in Figure 1, **Person** class holds the name, surname, birth date, and gender information about a family member. It also has two references to the Relation class; the **parents** variable and the **relations** array. Similarly, the Relation class has references to two Person objects, **p1** and **p2**, a relation name that represents the nature of the relationship, and a list of

Person objects that refer to children of this relation. Briefly, the variable **"parents"** points to the Relation object that holds the references to parent Person objects. The **relations** variable holds the relations of the Person object.

For example, let's assume that the user creates a new **Person** object.

```
john=Person('John', 'Connor', '28.02.1985', 'M', parents = null, relations = empty);
```

This object can stay on its own until John's parents are defined. Let's add his mother and his father:

```
sarah=Person('Sarah', 'Connor', '12.06.1964', 'F', parents = null, relations = empty);
kyle=Person('Kyle', 'Reese', '24.08.2004', 'M', parents = null, relations = empty);
```

In this scenario, Sarah and Kyle have a relationship; so these two objects are set as references in a Relation object, **r**=Relation(**sarah**, **kyle**), which sets **p1=sarah**, and **p2=kyle**. The child, John, should be in the **children** list (or array), so it should be added as **r.children.add(john)**. The new relation object should be set as the parents of john; **john.parents = r**;

```
r=Relation(sarah, kyle);
john.parents = r;
```

Other members of the family can be added to grow the tree using this approach. The Person and Relation objects that have been created should be kept in two arrays, so that they can be accessed without the need of their variable names. They are wrapped under the **FamilyTree** class, so that each family member and their relations can be accessed.

These classes and relationships between these classes meet **Functional Requirement 1, Functional Requirement 2** and **Functional Requirement 3** that have been defined in the requirements document.

Functional Requirement 1: The user shall be able to add a family member.
Functional Requirement 2: The user shall be able to define relationships between two family members that have already been added.
Functional Requirement 3: The user shall be able to add children to the defined relationships between two family members.

2.2 Behavioural Design

While the structural design of the classes can store the family tree in memory, the following methods are required to add functionality to the software.

Searching Family Tree for Members

Functional Requirement 4 states that the users shall be able to search for a family member. The FamilyTree class stores all family members in **members** variable, which is an array of Person objects. This list can be linearly searched for a specific Person. However, this requires the

Person class to provide a method to compare two Person objects. This could be provided by either a public **isEqual** method, or by overloading the equality (==) operator. The **isEqual(Person p)** method will accept another Person object and will compare its name, surname and the birth date values to the current object. It will return **true** if they are all the same.

This functionality will meet Functional Requirement 4.

Functional Requirement 4: The user shall be able to search for a family member.

Persistent Storage of Family Trees

The family trees created by the software shall be saved to and read from the disk. This allows the user to store the family tree so that they can continue updating at a later time.

The data for the family members can easily be serialized. A FamilyTree object can be serialized and can be streamed to a file. Later, it would be possible to deserialize it so that it could be loaded back to the software. To achieve this, the FamilyTree class should implement the **Serializable** interface. While the **Save** method will be a public function, **Load** should be a static public function so that it could be used without creating a FamilyTree object first. Both methods will require a **Path** to the file that will be operated on.

This functionality will meet Functional Requirements 6 and 7.

Functional Requirement 6: The system shall be able to save the current family tree to a file on the file system.

Functional Requirement 7: The system shall be able to load a family tree from a previously saved file on the file system.

Merging Family Trees

Although the software will operate on a single family tree while it is running, it shall also be able to merge two family trees. To merge two trees, one of the trees must be opened in the software first. While viewing this tree, another tree should be loaded using the **Load** method, but instead of replacing the current tree, it should merge them.

The merging is done as follows. We refer to the current tree loaded in the software as A, and the tree loaded with the **Load** method as B. Each family member in the trees can be iterated using the **members** array. A search is done on tree A for each **member** in tree B using the **isEquals** method. If **isEquals** returns **true** for a Person object **m** in Tree B and a Person object **n** in Tree A, then the reference to **m** is updated with **n**. This update should be repeated on the Relation objects in tree B, too. That is, if there is a reference to **m** in the **relations** array in tree B, each should be updated so that they refer to **n.** This approach updates all references to the same family members in both trees. If there are more than one same member, all will be updated with this approach. If there are no same members, then two trees will be merged without a connection.

This functionality will meet Functional Requirement 8.

Functional Requirement 8: The system shall be able to merge two family trees.

Finding Relations

The software shall be able to find the relation between two members of the family tree. The relationships are coupled, for example, anne-oğul, dayı-yeğen, or dede-torun. Therefore, finding one of them implies the other. The application will not define the relationship indirectly as "teyzemin kuzeninin yeğeni," but will be looking for direct relationships as in teyze-yeğen.

For direct relationships, the **shortest** path between two members must be found like in a graph. The class structure of the family tree, as shown in Figure 1, is essentially a graph. The Person class corresponds to the nodes in a graph. The edges are defined by the **parents** variable and **relations** array. For the "John Connor" example given in Section 2.1, John Connor is a node, and its neighbours are "Sarah Connor" and "Kyle Reese." The FamilyTree class is the one that holds all nodes and edges.

There are several algorithms to find the shortest path between two nodes in a graph. Dijkstra's algorithm is a commonly used one. Basically it finds the shortest path between all nodes in a graph, however, in most of its implementations there is a **source** and a **destination**, and the algorithm stops once the destination node is reached.

A **FindRelation(Person A, Person B)** method should be defined in the FamilyTree class to find the direct relationship between two Person objects. Since the relationships are coupled, finding the path from Person A to Person B would suffice. Given two nodes in a graph represented by the member variables in the FamilyTree class, Dijkstra's Algorithm will be implemented to find the shortest path between them. Each edge in the path will be used to reach a previously defined relationship. For example,

A -> parent -> child -> B : kardeş-kardeş

A -> parent -> B : anne or baba (depending on the gender of B) - oğul or kız (depending on the gender of A)

A -> parent -> parent -> B : dede or babaanne/anneanne (depending on the gender of B, and the gender of A's parent).

A -> parent -> child -> child -> B: amca/dayı or hala/teyze - yeğen (depending on the genders)

These examples show us that the paths are either parent, because of the **parent** variable, or child, because of the **relations** variable in the Person class. The genders play an important role, so the genders should be added to the path types.

These paths must be defined as direct relations, and the resulting paths must be compared to a list of paths to check if it matches one of these relations. Only one direction of the paths would be enough, the method could check the reverse path simply by changing parent edges to child, and child edges to parent. The method shall return the matched relation.

Since the relation between two members is not known at the beginning of the query, it is possible that there is no direct relation. It is also possible that there is no path from source to destination. In all these cases, the response should be "there is no direct relationship between Person x and Person y."

This functionality will meet Functional Requirement 5.

Functional Requirement 5: The user shall be able to query the relationship between two family members.

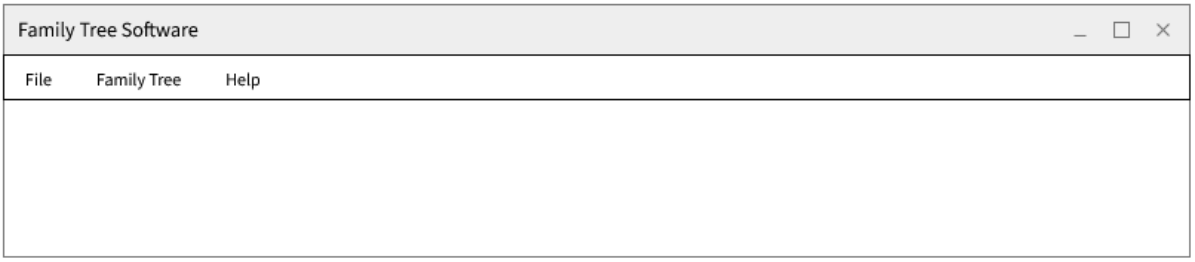
3 Graphical User Interface

The underlying classes and their methods provide a solid groundwork for the operations on the Family Tree. The graphical user interface shall provide means to enter and edit information on the family tree, as well as displaying it.

The user interface is planned to follow modern user interface guidelines. It will have a menu bar with the following actions:

- File
  - New
  - Open
  - Save
  - Save As
  - Exit
- Family Tree
  - Add New Member
  - Add New Relation
  - Merge Trees
- Help
  - Help
  - About Family Tree

There will be a toolbar with the following actions: New, Open, Save, Add New Member, Add New Relation, Help. As noticed, these actions are already present in the menu, but the toolbar will provide an alternative access.



The user interface is planned to be straightforward. For example, the dialog window to add a new member is designed to be as follows. It is possible of course to update the design during implementation.

As the menu items show, there will be a Help document for the project. The user shall be able to use the software after reading this manual. It will be accessible within the software, it will not be online. This will meet Non-functional Requirement 1.

Non-functional Requirement 1: The user shall be able to use the system after reading the manual.