**Abstract data types**

**1 – Setting data types**

In my program I want to implement reading settings from the text file, writing them there, and editing them. That's why I need class Setting, which will contain data about each setting, and class Settings, that will contain information about all the settings.

I will start by writing about class Setting. It must contain such attributes: setting\_name (str), possible\_values (list or function), current\_value (str), description (str), and possible\_values\_str (NoneType or str). The problem may occur if I want, for example, to create a setting values of which can be integer numbers bigger than some number. If I try to define such values using list, string representation of a Setting will be too big. That's why I added possible\_values\_str just for that case – to give shorter string representation of possible values for a setting. Possible values of the setting can be set either as list or as a function. The function returns True if value is in possible values for a setting. For example, if I want anything that ends with ".txt" to be a possible value, I will use function, and when I need to choose between "yes" and "no", I will use list

Methods of Setting class are pretty standard. Because all implementation is encapsulated, they are used to get all the information from the class example (get\_setting\_name(), get\_current\_value(), get\_descrption(), set\_value(current\_value), \_\_str\_\_() (for showing on screen), \_\_repr\_\_() (for writing into a file)). Some of the protected methods will be \_set\_possible\_values(possible\_values, possible\_values\_str) (static), which will return tuple with function for checking current value and string representation of possible values.

Settings class has list of all settings as an attribute (private). There must be methods for changing values of the settings anyway (for easiness). Methods \_\_str\_\_() and \_\_repr\_\_ are for outputting information about the settings on the screen and to the file respectively. Method read\_file must be for changing settings.

**2 – Data (Friends)**

I want to represent this type of data as a linked tree. But this won't be a standard tree with standard methods. Because list of friends is a part of a user account, it would be easier for me to create a User class, which would be a tree and a user at the same time. Attributes: parent, list of children, value which is True if the user already was upper in the tree. Methods (I won't explain self-explanatory ones): get\_parent(), get\_id\_num(), get\_children(), get\_root() – return root of the tree the User is in, is\_root() – return True if current user is a root of a tree, all\_users – return list of all nodes, which are children, and (grand \* n)-children of current node, height() – return height of a tree with current user as a root, get\_level() – return level of a current user in the tree (level of the root is 0, of its children – 1 , of their children – 2, and ....), users\_of\_level(level\_num, ignore\_repeated) – return all users with level level\_num in tree with current user as a root (ignore\_repeated indicates whether the program should ignore repeated users), first\_occur() – return first occurrence of self in tree with root as main root, user\_by\_id() – search tree for User by id number, and return User object or None, set\_parent(parent), set\_children(list\_of\_children), \_\_str\_\_(), \_\_repr\_\_(), in\_file\_user() – return in-file representation of a User, in\_file\_tree() – return in-file representation of an entire tree with current user as root.

If the user already is in a tree, but still is added (it is possible), there is an attribute in each node (tree) which indicates if this happened. If it is, node has this value set to True. But friends of User with that attribute are still found for purpose of making tree more balanced and easy to use.

Various methods must have an attribute which indicates if I want to include all nodes that describe the same user. For example, for users\_of\_level method. This will be useful for building graphs later on.

Each user is described by id number here, not by username.

**Short information:**

**Setting:**

Attributes:

* \_setting\_name – str
* \_possible\_values\_func - function
* \_possible\_values\_str – str
* \_current\_value – str
* \_description - str

Methods:

* get\_setting\_name(self) - str
* get\_current\_value(self) - str
* get\_description(self) - str
* \_\_str\_\_(self) - str
* \_\_repr\_\_(self) - str
* \_set\_possible\_values(possible\_values, possible\_vaues\_str) - tuple

**Settings:**

Attributes:

* all\_settings – list

Methods:

* setting\_by\_name(self, setting\_name) – Setting
* read\_file(self, file\_name) – None
* write\_into\_file(self, file\_name) – None
* \_\_str\_\_(self) - str

**User:**

Attributes:

* \_id\_num - int
* \_parent – User or None
* \_children – list or None
* already\_was\_in\_tree – bool

Methods:

* get\_parent(self) – User
* get\_id\_num(self) – int
* get\_children(self) – list
* get\_root(self) – User
* is\_root(self) – bool
* all\_users(self, ignore\_repeated=True) – list
* height(self) – int
* get\_level(self) – int
* users\_of\_level(self, level\_num, ignore\_repeated=True) – list
* first\_occur(self) – User
* user\_by\_id(self, id\_num) – User
* set\_parent(self) – None
* set\_children(self) – None
* \_\_str\_\_(self) – str
* \_\_repr\_\_(self) – str
* in\_file\_user(self) – str
* in\_file\_tree(self) - str