Program #1 Design

Written by Alexander DuPree

Task and Purpose

Program #1's focus is in the design, implementation and manipulation of a **linear linked list** (*LLL*) data structure to store and manage an **abstract data type** (*ADT*). As such, we were tasked to perform the following:

- Create a *LLL* that will store a unique set of *categories* (e.g., assignments, readings, labs)
- Each *category* contains a *LLL* that stores a unique set of *projects*.
    - A *project* is an item of work belonging to a specific category (e.g., reading #1 belongs to readings *category*)
- The client should have the following functionality:
    1. Add a new *category* of work.
    2. Remove a *category* of work.
    3. Add a *project* within a *category*
    4. Remove a *project* from a *category*
    5. Display all *categories*
    6. Display all *projects* in a *cotegory*

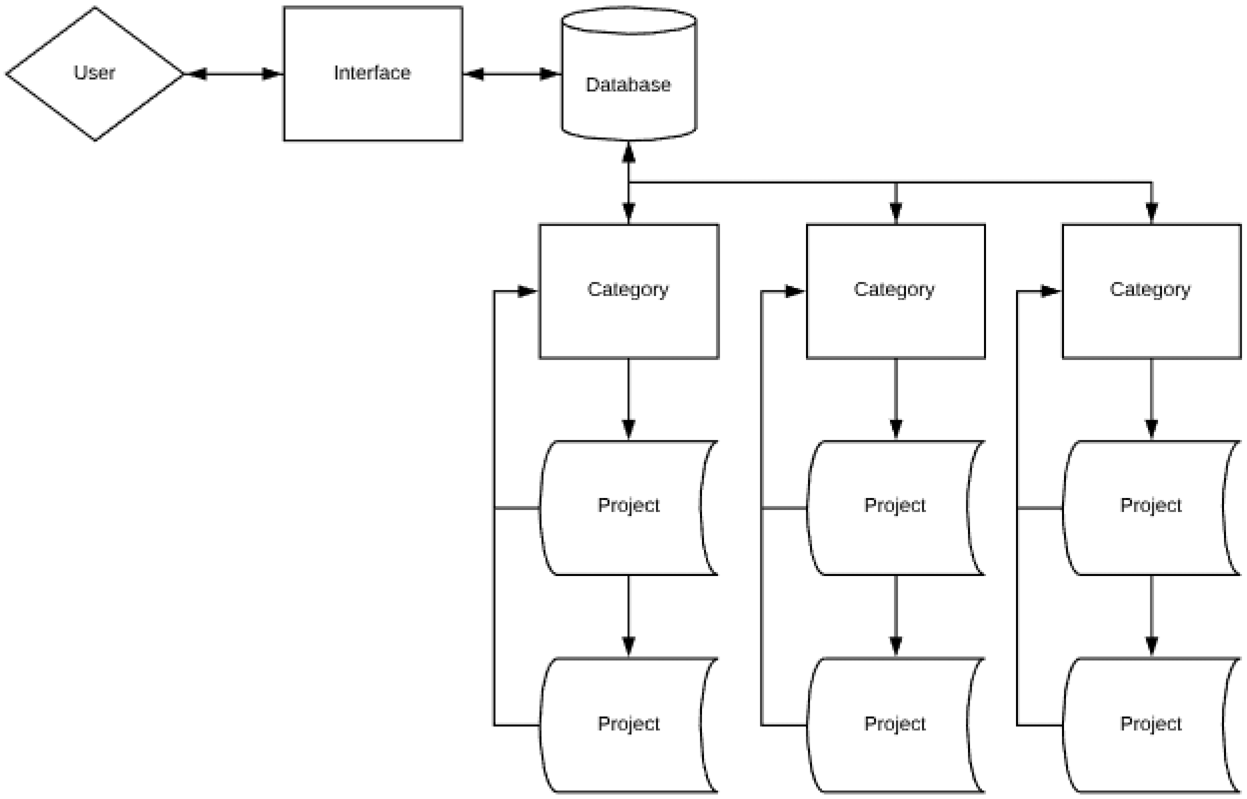
Design Considerations

One of the primary goals of this project is to create an *ADT* that "hides" the implementation of our *LLL* data structure. It is also important to design the *LLL* and *ADT's* to be "plugs" and "sockets". Meaning that even though the *ADT* is dependent on a *container*, the exact implementation of that container should be abstract an detail. This will allow future maintainers of this program to remove the *container* from it's "socket" and "plug" it with a better performing data structure.

As for as the client-side programmer, he/she should only have knowledge of how to use the *ADT's* public interface to create a user-application. Each *ADT* should perform as a 'black box', accepting requests and returning data as necesssary.

Data Flow

With this design consideration in mind, the data will flow simliar to this



- The user interacts with the interface generating *requests* to add/remove/display a *category* of work or to add/remove/display a *project* within a *category*.
- The interface validates, cleans, and packages the user's input into a format the *database* will accept.
- The *database* will execute the *request* by performing one of the following actions:
  1. Create a *category* of work.
  2. Remove a *category* of work.
  3. Request *project* information from a *category*.
- After the request has been executed the *database* returns either the requested information or a confirmation receipt.
- The interace presents the data in a formatted fashion then listens for user input.

Data Structures and Abstract Data Types

The scope of this assignment required the liberal use of the **linear linked list** data structure to perform insertion, deletion, and find operations. The *Database* and *Category ADT's* will act as wrapper classes for the *LLL* data structure; utilizing the *LLL* to perform operations within the it's scope of responsibility.

The following is a summary of each *ADT* and Data structure present in this program:

Sorted_List

The *sorted list* is the primary data structure for this program. The *sorted_list* is a **linear linked list** with only one public modifier: *add()*. This modifier will insert **Unique** data into the list at a **Sorted** position. Keeping the list in a sorted state will prevent from having to order the list later. This also satisfies another assignment requirement to keep the *category* types sorted alphabetically.

The most imporant feature of the *sorted_list* structure is that it is fully-templated. This feature is to accomadate the requisite to create a *LLL* that holds *category* objects, and have each *category* hold a *LLL* of *projects*. Because of this requisite there seemed to be two options: 1. Write the *LLL* container to support a *project ADT* and then duplicate that code; replacing any reference to a *project ADT* with a *category ADT*.

or

1. write the code once as a template and be done with it.

Alternatively, an abstract base class could have been created for the *category* and *project ADT's* and then they could have been passed around polymorphically; however, I felt the templated approach to be the most suitable.

The following is a summary of the *sorted_list* public interface:

Function	Summary	Parameters	Return
add	inserts **UNIQUE** data into a **sorted** postion in the list and returns true.	A read-only reference to the data to be inserted	Returns true if the insertion was successful
begin	returns a **read only** iterator to the beginning of the list	None	If the list is empty, the begin iterator will be equal to the end iterator
end	returns an iterator to the element "one-past" the end of the list	None	Dereferencing an end iterator causes undefined behavior
size	Returns the number of elements in the list	None	The sorted_list contains an internal counter so that calling size() is a constant time operation
empty	Tests if the list is empty	None	Returns true if the list is empty
clear	Delete each node in the list	None	Recursively travels the list, deleting each node as the stack unwinds
Relational Comparisons	Only the == and != operators are overloaded	A read-only reference to a list containing the same data type	Returns false if the sizes are different, then compares each element of the both lists returning false on any inequality.

Project

The *Project ADT* stores attribute data relating to a specific *category* of work. For example:

- If an Assignment *category* was created then its fields could be:
  1. Name
  2. Due Date
  3. Due Time

4. Late Date
5. Data Structure

- Each *project* will contain the requisite field data in the form of a string.

The *Project ADT* stores this data sequentially in a vector object. The *category* class maps the specific data from the vector to the correct field. This allows a *project* to be instantiated to support any number of *categories* and a variable number of fields. The only requisite field a *project* must have is the "Name" field. The "Name" field acts as the unique identifier for the *project* allowing it to be compared against and searched for.

The following is a summary of the *project* public interface:

Function	Summary	Parameters	Return
name	Returns the name identifier	None	Returns a read-only reference to a custom string class object (SString)
begin	Returns a **read only** iterator to the first string of data	None	If the project is default constructed, the begin iterator will be equal to the end iterator
end	Returns an iterator to the element "one-past" the end of the list	None	Dereferencing an end interator causes undefined behavior
Relational Comparisons	The equality and comparison operators are both overloaded	A read-only reference to another *Project* type	Uses the *Project's* name field as the basis of comparison

Category

The *category ADT* will act as a wrapper class for our *sorted_list LLL* data structure. It will provide methods to add, remove, and inspect *Project ADT's* stored in it's *LLL*.

Below is a list of the public functions the *database* will interact with.