MIDTERM PROJECT

Assignment 5

Team members:

Ben Halpern

Itz Rodriquez

Tony Cheng

Jose Chavez

Thien Nguyen

Jesus Sierra

Santa Ana Community College

CMPR 131

# 

[Executive Summary: 3](#_Toc85339984)

[SDLC Flow and plan: 3](#_Toc85339985)

[Specification: 4](#_Toc85339986)

[Roles and Responsibilities: 4](#_Toc85339987)

[Goals and objectives: 5](#_Toc85339988)

[Design Decisions: 5](#_Toc85339989)

[Functional Description: 5](#_Toc85339990)

[Design Objectives: 5](#_Toc85339991)

[FlowChart/ Program Hierarchy: 7](#_Toc85339992)

[UMLs: 11](#_Toc85339993)

[Solution Details: 12](#_Toc85339994)

[Solution Overview: 12](#_Toc85339995)

[Main Components: 12](#_Toc85339996)

[Classes: 12](#_Toc85339997)

[Student: 12](#_Toc85339998)

[ListDriver: 13](#_Toc85339999)

[VectorDriver: 13](#_Toc85340000)

[IntContainer: 14](#_Toc85340001)

[Source Code Documentation: 14](#_Toc85340002)

[Design Patterns: 14](#_Toc85340003)

[Coding Standards: 14](#_Toc85340004)

[Coding Naming Convention: 14](#_Toc85340005)

[Algorithms: 15](#_Toc85340006)

[Testing Documentation: 15](#_Toc85340007)

[Testing Plan: 15](#_Toc85340008)

[Test Cases: 15](#_Toc85340009)

[Test Results: 15](#_Toc85340010)

[Sources: 16](#_Toc85340011)

[Glossary: 17](#_Toc85340012)

[Terms: 17](#_Toc85340013)

[Coding Convention Reference: 17](#_Toc85340014)

[Naming Conventions: 17](#_Toc85340015)

[Condition Documentation: 17](#_Toc85340016)

[Class Invariants: 18](#_Toc85340017)

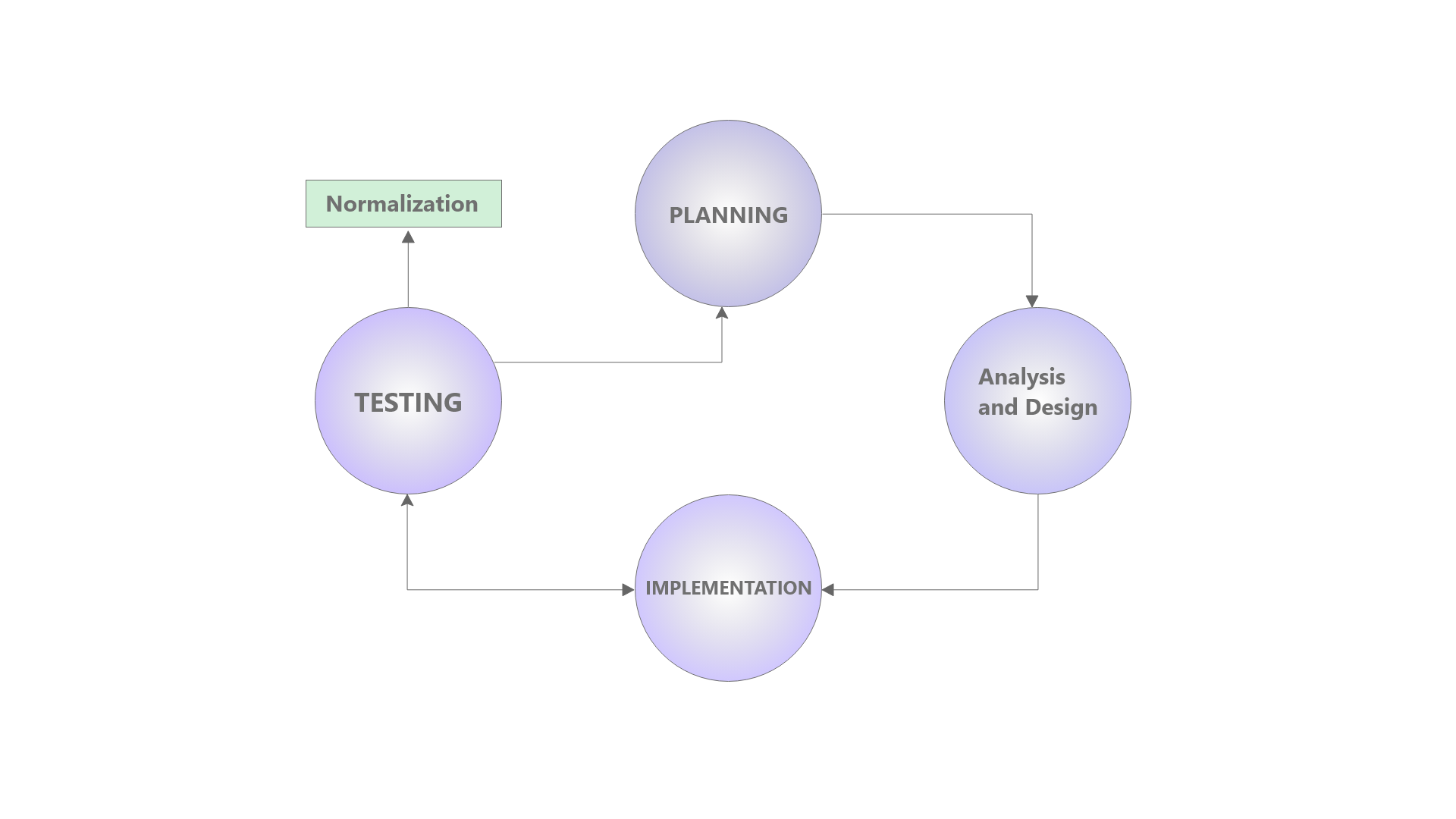
[Hierarchy Charts: 18](#_Toc85340018)

[UMLs 18](#_Toc85340019)

# 

# Executive Summary:

## SDLC Flow and plan:



The *Software Development Life Cycle (SDLC)*[[1]](#footnote-1) follows the sequence shown above. Starting at the planning stage of the software development life cycle, the programming team takes the *specification*[[2]](#footnote-2) given for the program and breaks it apart into smaller digestible segments. The assigned parts, the small digestible segments, are then analyzed and further broken down in the *Analysis and Design[[3]](#footnote-3)* phases. During this stage of the life cycle, the specification is broken down into a program flow, to understand the flow of the overall program. Each digestible stage follows its own iteration of the life cycle, branching back into the main flow of the program as each task and subtask are completed. In the analysis and design phases of the life cycle, each task, that was broken down into components and the algorithms, are tested in the efforts to optimize them. The different *implementations[[4]](#footnote-4)* and components[[5]](#footnote-5) of the program are mapped out, in forms such as *hierarchy charts[[6]](#footnote-6)*, *pseudo code[[7]](#footnote-7)*, *flow charts[[8]](#footnote-8)* and more *logical design elements[[9]](#footnote-9)*. Implementing these designs in the code and testing the code after each new addition, looping through the life cycle until the entire program is complete. Once the program is complete the life cycle continues as the code evolves and is maintained.

Tasked with programming and application to utilize the list, and vector standard Template library classes in an application running on a Student object as the underlying data type and a final application demonstration running on an integer data type. Driven by a menu function prompting the user for various options through the various tiers of the application. We split up the work in an agile based approach, the programming that implemented the list and vector class was done simultaneously, then merging those methods taken and learned from the implementation into the application demonstration option. Testing each method after it was written, basing each method of the UML and updating the UMLs for each class as the programming progressed. Coordinating regularly, at a rate of meeting every other day, to divide up the remaining work at each completed stage following the Development Life Cycle.

# Specification:

Given an example executable, we used the example executable as a reference for the style of the user interface of the program, and reverse engineering the program in order to understand the algorithms and methods within it. Following the specification given, we had to also debug the executable file given to us as it was not without *logic errors[[10]](#footnote-10)*.

We were specified to create a menu-driven program that utilizes the Vector and List classes from the Standard Template Library using the three options shown in the executable. These three options were a List container option, a Vector container option, and an Application option which uses a List and or Vector. Each of these options were broken down into sub menu-driven options demonstrating the various member functions of the List and Vector classes.

## Roles and Responsibilities:

Student class: Ben, Itz

Private members of the class would be defined by both team members to be familiarized with them. Accessors would be coded by one group member and the mutators would be coded by the other team member. The team member One team member would write the relational operators, the other would write the input and output stream operators.

Vector class option: Jesus, Jose, Tony, Itz

List class option: Thien, Ben

Application class option: Jesus, Tony, Jose

## Goals and objectives:

Upon analyzing the design, we noticed that two options of the program relied on a Student class. Starting the code on the Student class was a priority, since other classes depended on them. Once the Student class was finished then a more *agile[[11]](#footnote-11)* approach was able to be implemented and the vector section and list section of the code could be completed simultaneously, along with the Application section of the project. Using the first two sections of the program to best familiarize ourselves with the *Standard Template Library (STL)[[12]](#footnote-12)*, we were able to better implement the methods we designed that were efficient and logical for the Application class section.

Objectives:

## 

# Design Decisions:

With the specification in mind, we designed the utilization of the STL methods of the Vector and List class in our menu-driven classes, to utilize code reusability. We designed methods using iterators of private STL class objects stored within each menu-driven class, to traverse elements within each container. Many overloaded operators were designed in the underlying student class to minimize the coding overhead in the other menu-options.

## Functional Description:

The Student class is designed to hold demographic information about the student. The information of the student is also designed to be retrieved and outputted. Student class has many operator overloads to facilitate in comparing private members of two Student objects, assigning values of one Student object to the other, inputting via file and outputting.

The ListDriver class is designed to work ...

The VectorDriver class is designed to work ...

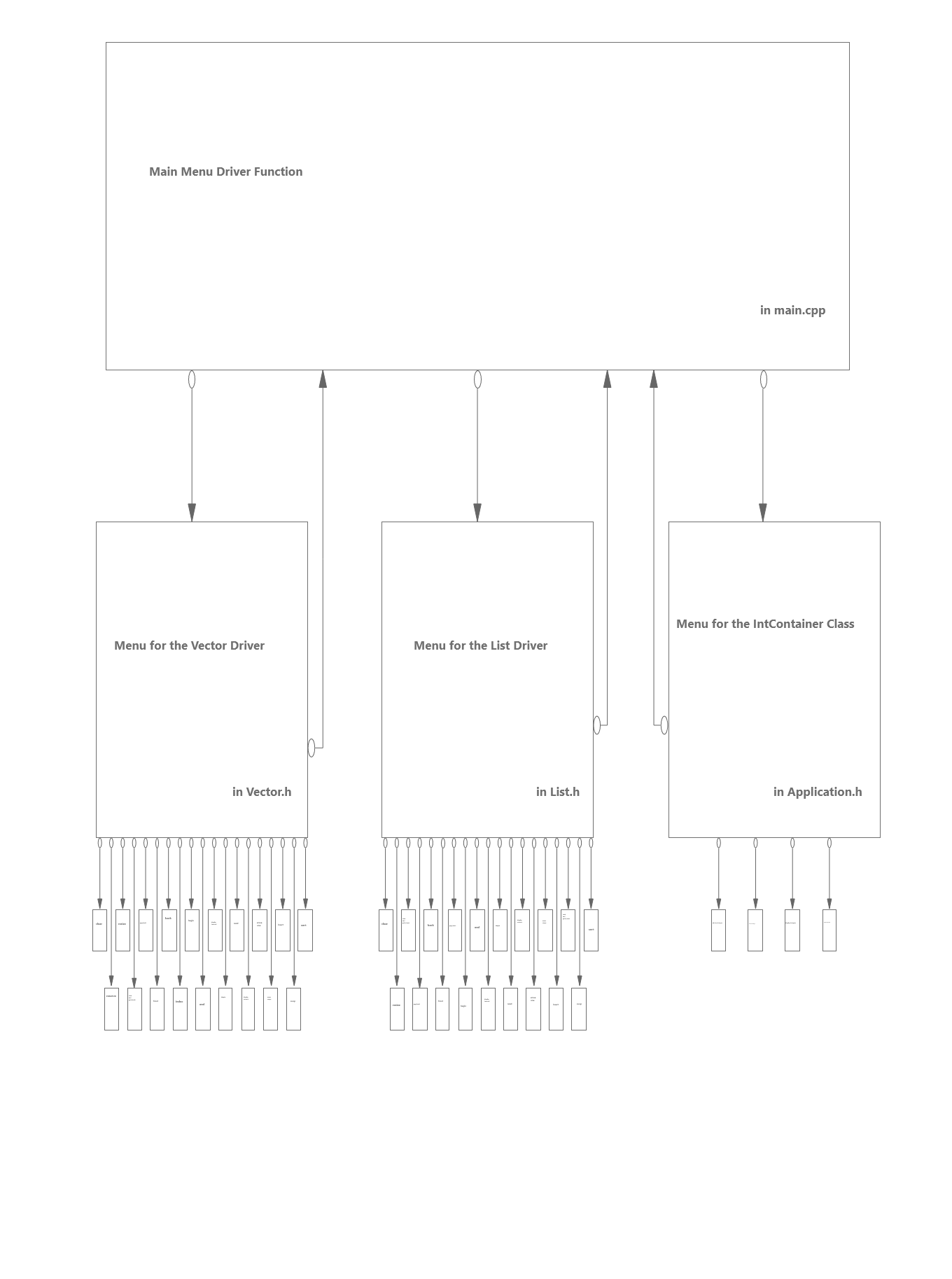
## 

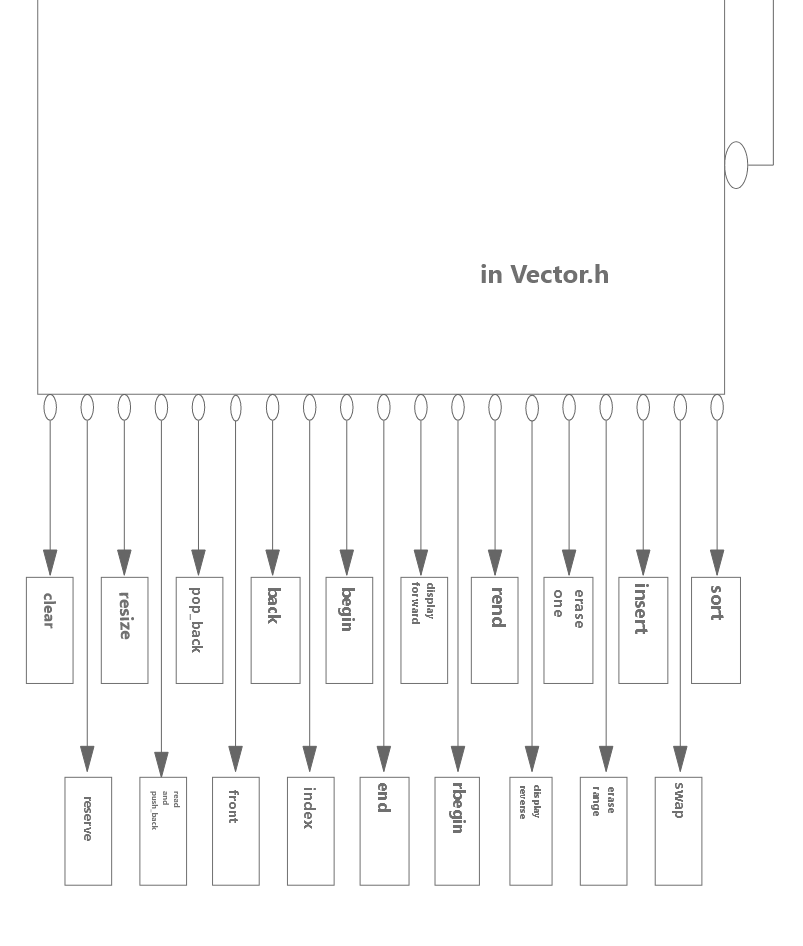
## Design Objectives:

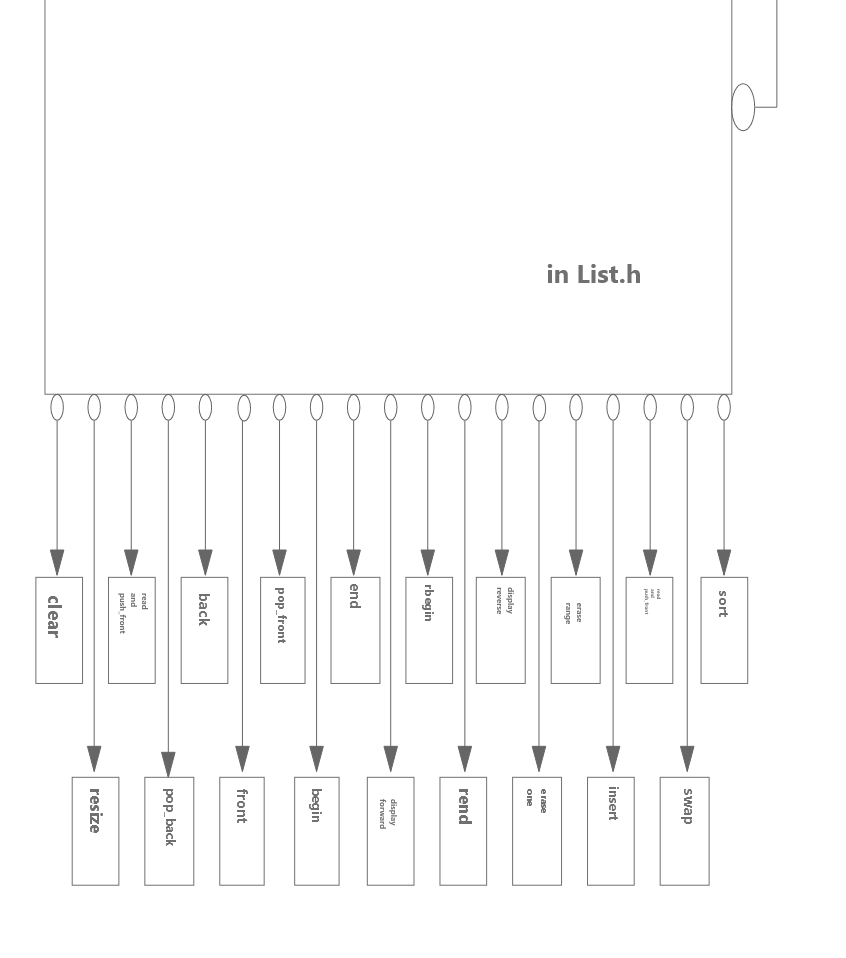
Our objectives in designing the program are directed to optimize the transversal and manipulations of the Vector and List STL classes to that of the given specifications. Designing the classes with self-containment in mind to allow for both code reusability and ease of use when implementing the meu-driven classes. In addition, our objectives are aligned with displaying and demonstrating our understanding of the Standard Template Library classes used during the reverse engineering of the executable given during the specification.

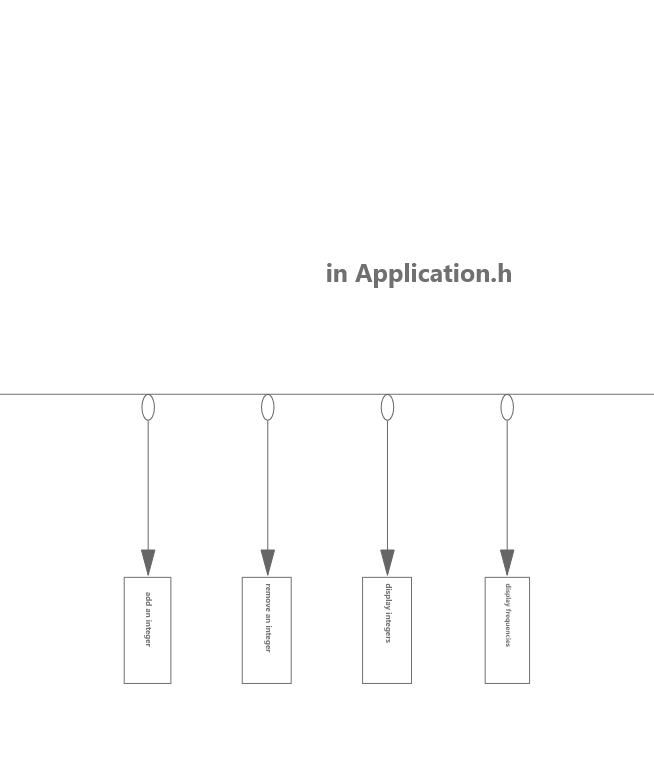
## FlowChart/ Program Hierarchy:

See Glossary (pg. x … y )









## UMLs:

See Glossary (pg. x … y )

### 

# Solution Details:

what we considered during implementing our design to create our solution.

## Solution Overview:

Overview of what our solution does.

### Main Components:

### Classes:

* Student.h
* List
* Vector
* Application

#### Student:

Private member *mName*, holds a string that will hold a student’s first and last name. Private member *mLEVELS\_ARRSIZE*, is a constant integer of a value of four,which is used to establish the size of private member, *mLEVELS*; *mLEVELS* is a constant string array with 4 components, “Freshman”, “Sophomore”, “Junior” and “Senior”. These *mLEVELS* represent the four different academic student tiers. Each student must have a level associated with them. The private member *mGPA* holds a double value, representing the student’s grade point average. Next the private member *mEmpty* holds a boolean that is set to the value of true initially. The final private member is *mDEBUG* which is a constant boolean initially set to the value of false. *mDEBUG* is used for debugging purposes to print out the status of the code as it’s running.

Default constructor sets the *mName* to “unknown”, the *mLevel* to zero and the *mGPA* to zero. Once the object is initialized with no parameters, these values are passed in. The copy constructor passes in a student object as a parameter and sets those values to the other object. The *mGPA*, *mLevel*, *mName*, and *mEmpty* values are passed into the new student object.

The *getName* is a constant accessor function and will return the value of the private member *mName*. The student’s whole name would be retrieved from this function. The *getLevel* accessor is a constant accessor that will return a string data type of the *mLEVELS* array value. Using a case switch case it will retrieve the *mLEVELS* array string value relative to the *mLevel* integer value, this will retrieve the school tier level. *getNLevel* is an accessor that will return the integer value of the *mLevel*. This retrieves the value of the integer that can be used to reference the index value of the *mLEVELS* array. *getGpa* is an accessor that will return a constant double. The *mGpa* will be needed to be retrieved for outputs in the program. The function *empty* is a constant accessor boolean function that returns the value *mEmpty*. This value is used to check if there’s a *mName* and a *mGpa* initialized. The function *error* returns a constant boolean, which retrieves the value of *mError*. This is used to see if there is an error with the program, with the boolean error set to true when an error occurs and a student value is returned.

The *setName* function is an overloaded function with two versions, one version does not take in any parameters it will prompt a message to add a string value that will assign it to *mName*. The other version will take in a string value as a parameter and assign it to *mName*. The function *setLevel* is an overloaded function with three versions, one version does not take in any parameters it will prompt a message to add an integer value that will assign it to *mLevel*. Another version will take in an integer value as a parameter and assign it to *mLevel*, and will check if the value is within the range of 0-4 it will assign the value to *mLevel*. Another version a string value is passed as a parameter and assigns the corresponding integer index value that equals the string value in the string array of *mLEVELS*. The *setGpa* function has two different versions, one version with no parameters and it prompts a message to enter a double within the range then it will assign that value to the *mGpa*. The other version will pass in a double value as a parameter and will assign it to *mGpa* if it’s within range of 0-5.0. All of these functions will call *mCheckEmpty* which checks if the container is empty.

The overloaded operators have constant and non constant versions, due to functions calling on the operator, some may need one version or the other. The ***operator******==*** is an overloaded operator that returns true if the *mGpa*, *mLevel* and *mName* of the two objects being compared are the same, if either of those values are not equal then it will return false. This operator is checking for if two objects are actually the same student. The ***operator >=*** is an overloaded operator that returns true if the *mGpa*, *mLevel* and *mName* of the object on the left is greater than or equal to on the right, if either of those values are less than or not equal to the object on the right then it will return false. The ***operator******<=*** is an overloaded operator that returns true if the *mGpa*, *mLevel* and *mName* of the object is greater than or equal to on the right, if either of those values are greater than or not equal to the object on the right then it will return false. The ***operator******<*** is an overloaded operator that returns true if the *mGpa*, *mLevel* and *mName* of the object is less than to the object on the right, if either of those values are greater than the object on the right then it will return false. The ***operator******>*** is an overloaded operator that returns true if the *mGpa*, *mLevel* and *mName* of the object is greater than to the object on the right, if either of those values are less than the object on the right then it will return false. These four operators are used for comparing students’ values and sorting them. The ***operator =*** is the assignment operator will appoint the values from the object on the left to the objects on the right. The values of *mGpa*, *mName*, *mLevel* and *mEmpty* from the object on the left will equal to the object on the right.

The ***operator <<*** is a friend function that will output the *mName*, the level as a string value and the *mGpa*. This is used to output the information of the student for the user to review. This friend function has two versions: the constant and non constant. The two versions are needed for both versions of methods calling on this operator. The ***operator******>>*** has two versions, one is used for input from a file stream and a non-file stream. The operator function will call on the mutators to set the *mName*, *mLevel* and *mGpa*.

#### ListDriver:

Private member *mListOne* is a list container that stores student objects and is the only private member variable. The constructor *mListDriver* will be called to initialize a list to be manipulated.

#### VectorDriver:

Private member *mMyVector* is a vector container that stores student objects. The default constructor *VectorDriver* will initialize the default vector.

The *getSizeOfVector* is an accessor that will return an integer value that is the size of the vector. This is used to see if the vector is empty for verification if the container has elements.

#### IntContainer:

# Source Code Documentation:

A best practice when programming is adhering to a set of standards and patterns when coding and when coming up with names for classes, variables, and various function types. The standards we used in our solution, which we implemented throughout and at the end during the *normalization[[13]](#footnote-13)* of our code, are outlined in the following sections.

## Design Patterns:

At the header of each file, we placed header comments identifying the file name, the team members, and other identifying information about each file including a description. We organized the classes, using the structural pattern placing the accessors first followed by the mutator functions all after the constructor definitions inline in the classes. Placing navigating comments throughout the classes to help identify each section within the class. Some classes have an overloaded constant member version in order to provide access to the member within other constant defined member functions.

## Coding Standards:

We attempted to adhere to coding standards that attempt to mimic those of the Standard Template Library in order to better fit implementation with other classes within the STL to allow easy usability with those other classes.

## Coding Naming Convention:

When naming classes, we capitalized the first letter of the class in *Camel Hump[[14]](#footnote-14)* format, and made private members of the class begin with the prefix ‘m’ to easily identify it as a member variable. We then used the prefix ‘p’ to identify formal parameters passed to each function to identify them as parameters. For the ListDriver and the VectorDriver Classes the postfix “Driver” was placed after the underlying data container type used in the class in order to avoid confusion, the same was applied to the application section of the program which implements the IntContainer Class, in a similar self-explanatory convention. The public member functions follow camel hump format or mimic that of the STL container class functions which are called within it, to add a layer of abstraction to the underlying container.

Documenting our code with three forward slashes to identify and utilize InteliTips[[15]](#footnote-15) documentation within visual studio to better allow reading of documentation in Microsoft Visual Studio when implementing each function. Before each function, the precondition and postconditions are mapped and described. Short tag descriptions for commonly identifiable function traits are included within square brackets prior to the Precondition and Postcondition.

### Algorithms:

When looping we opted to use the for loops when traversing the lists and vectors, and used the while loops and do-while loops for menu driven portions of the code.

We implement the STL function member function *sort* of the vector and list classes, which uses the less than operator to implement a linear sort of the items within the container. The sort function sorts by creating a “strict weak ordering”[[16]](#footnote-16) of the elements within the containers, and sorting with respect to that order. In order to implement the sort function, the less than operator in the Student Class must be overloaded and meet the preconditions needed for using the sort function.

# Testing Documentation:

Describe a little about the testing phase of the report and what it is for and what it helped us see.

## Testing Plan:

After each member function and member variable is defined, we created separate test functions to properly exercise all the code, and test the functionality of the code as well. During this testing, we put an emphasis on the testing of the sanitization of all input within the code. We sanitized the code using an input header file which was provided to use for sanitization of input from the end-user. Once the code was fully combined into a single working program, we tested the limits of the functions and sanitization by testing known out of bound values and boundary values as well. Additionally, we added a debug flag, represented by the Boolean *mDEBUG* as a constant private member variable in each of the classes to easily turn on and off a verbose debug display when exercising the code.

## Test Cases:

Some test cases that we used in our code, as outlined above, we passed incorrect data types as actual arguments when exercising the code in order to see if the code would break or handle the invalid data type entry as desired. We also attempted to manipulate empty containers in unconventional manners, such as sorting an empty List or Vector, or displaying an empty List or Vector. We compared unusual results to the executable file provided to us during the specification.

## Test Results:

A bug that was present in our code which upon discovery was quickly remediated. This bug existed due to the comparison of two empty student objects producing an incompatible Boolean response for the comparison, which produced a runtime error, when the sort function created it’s strict weak order.

# Sources:

<https://gbksoft.com/blog/types-of-software-development-documentation/>

<https://www.altexsoft.com/blog/business/technical-documentation-in-software-development-types-best-practices-and-tools/>

<https://www.scrum.org/resources/what-is-scrum/>

<https://www.agilealliance.org/agile101/>

<https://www.cplusplus.com/reference/list/list/sort/>

# Glossary:

## Terms:

1. Software Development Life Cycle:
2. Specification:
3. Analysis and Design:
4. Implementation:
5. Components:
6. Hierarchy Chart:
7. Pseudo Code:
8. Flow Chart:
9. Logical Design Forms:
10. Logic Errors:
11. Agile Workflow:
12. Standard Template Library (STL)
13. Normalization:
14. Camel Hump Format
15. InteliTips:

## Coding Convention Reference:

### Naming Conventions:

[tags] are used to define design element comments such as this one and those of [const] for constant members and accessors, and [mutator] for mutator methods

When the tag is lower case the design element is a local comment, when the first letter is capitalized the design element is a global tag relating to a comment with a scope pertaining to the document as a whole.

The names of classes are capitalized following the CamelHump naming convention. 'm' is used as a prefix to defined private members of a class following the CamelHump naming convention. 'p' is used as a prefix to define formal parameters of methods following the CamelHump naming convention

### Condition Documentation:

Prior to functions, the documentation of the precondition and postcondition are described with three brackets so that the descriptions would work with Microsoft Visual Studio InteliSense, which produces would then display the precondition and postcondition when as a tip when the function is being implemented in the various parts of the program.

### Class Invariants:

Class invariants provide a list of the various methods in each class and provide a short description outlining the use and implementation notes for that class. The methods in the class invariant don't provide formal parameters but instead they show the datatype of each formal parameter in a similar format and style that might be seen in a prototype declaration.

Flow, Structural, and Document comments are seen throughout the code and are noted by the use of only two forward slashes '//' when defining the comments. A series of '###' may be used to denote flow indicator to help the programmer navigate the various classes

## 

## Hierarchy Charts:

## UMLs









1. Software Development Life Cycle: (Glossary pg) [↑](#footnote-ref-1)
2. Specification: (Glossary pg ) [↑](#footnote-ref-2)
3. Analysis and Design: (Glossary pg) [↑](#footnote-ref-3)
4. Implementation: (Glossary pg) [↑](#footnote-ref-4)
5. Components: (Glossary pg) [↑](#footnote-ref-5)
6. Hierarchy Chart: (Glossary Pg). [↑](#footnote-ref-6)
7. Pseudo Code: (Glossary pg). [↑](#footnote-ref-7)
8. Flow Chart: (Glossary pg). [↑](#footnote-ref-8)
9. Logical Design Forms: (Glossary pg). [↑](#footnote-ref-9)
10. Logic Errors: (glossary pg). [↑](#footnote-ref-10)
11. Agile Workflow: (Glossary pg). [↑](#footnote-ref-11)
12. Standard Template Library (STL): (Glossary Pg). [↑](#footnote-ref-12)
13. Normalization: (glossary pg). [↑](#footnote-ref-13)
14. Camel Hump Format: (glossary pg). [↑](#footnote-ref-14)
15. InteliTips: (Glossary pg). [↑](#footnote-ref-15)
16. https://www.cplusplus.com/reference/list/list/sort/ [↑](#footnote-ref-16)