

MIDTERM PROJECT

Assignment 5

Team members:

Tony Cheng

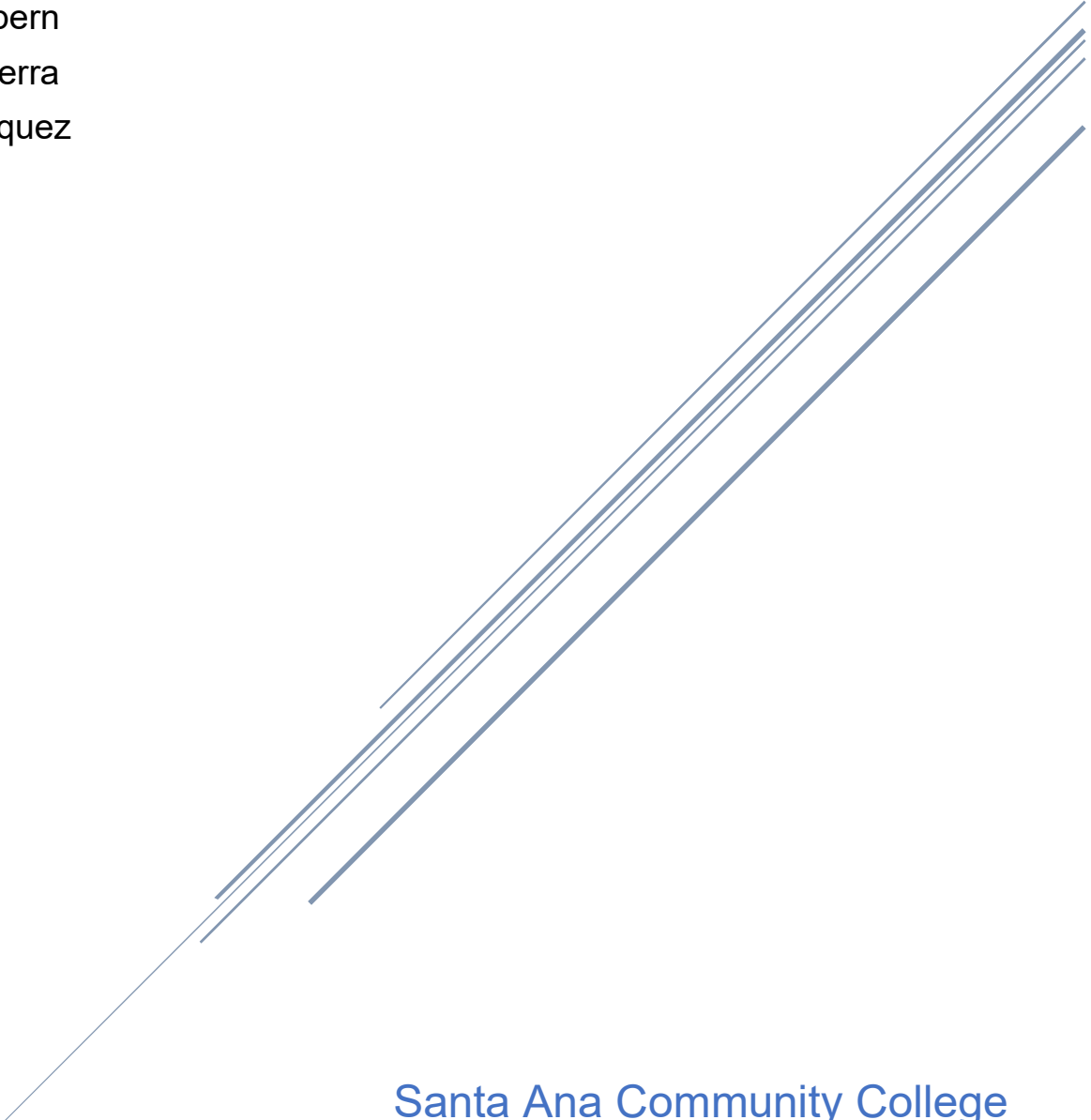
Jose Chavez

Thien Nguyen

Ben Halpern

Jesus Sierra

Itz Rodriquez



Santa Ana Community College
CMPR 131

Executive Summary:	3
SDLC Flow and plan:	4
Specification:	5
Roles and Responsibilities:	6
Student Class:	6
VectorDriver Class	6
ListDriver Class	6
IntContainer Class	6
Goals and objectives:	6
Design Decisions:	7
Functional Description:	7
Design Objectives:	8
Flow-Chart/ Program Hierarchy:	8
UMLs:	8
Solution Details:	9
Main Components:	9
Classes:	9
Student:	9
ListDriver:	10
VectorDriver:	12
IntContainer:	13
Source Code Documentation:	14
Design Patterns:	14
Coding Standards:	14
Coding Naming Convention:	14
Algorithms:	15
Testing Documentation:	15
Testing Plan:	15
Test Cases:	15
Test Results:	15
References	16
Glossary:	17

Terms:	17
Coding Convention Reference:	18
Naming Conventions:	18
Conditional Documentation:	18
Class Invariants:	18
Flow Charts:	19
Hierarchy Charts:	20
UMLs	23
Table of Figures:	26

Executive Summary:

Tasked with reverse engineering the program and examining and exercising our knowledge of the Standard Template Library, we created our program to look similar to that provided to us during specification. We attempted to retain many of the same naming conventions and mimicking function names of the STL methods called within in our naming conventions for methods of each class in order to improve readability and provide a level of abstraction¹. We were tasked to create three sections, each section running its own menu driven function, which each drove a different class. Two of the sections ran on a similar underlying data type, the Student Class, which had to be made prior to the completion of these sections. Adhering to an agile workflow after the Student Class was created, we outlined the flow of the main program and then broke the tasks up.

We first ran the example program and figured out the results of each possible choice. Also test how the program would react under specific situations that would normally lead to failure in the program and compared those situations to those in our code.

We created and implemented four separate classes, the Student Class, the VectorDriver Class, the ListDriver Class, and the IntContainer Class. Each class exists in its own header file with the driver function included within.

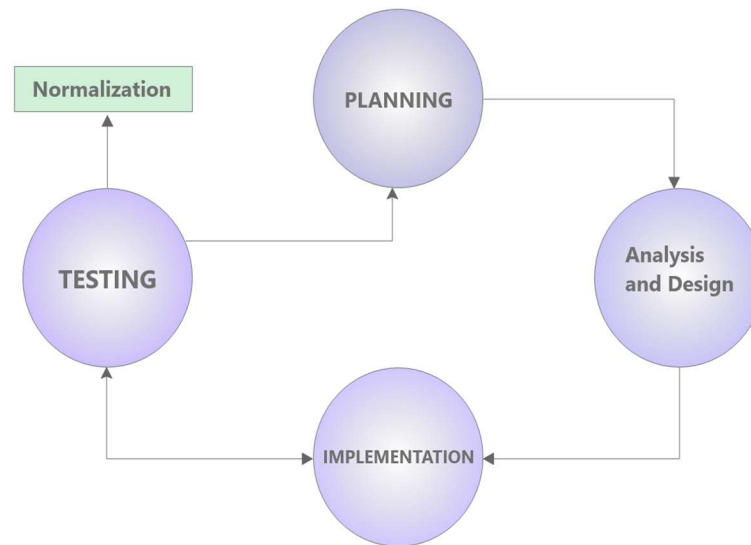
Coding reusability: We adhered to coding reusability rules by attempting to make the classes as self-contained as possible in the scope of the application that was specified to us. All the classes depend on the input header file, the iostream header, and those of the container of the Standard Template Libraries used. The VectorDriver and the ListDriver Classes additionally rely on the Student Class as the underlying data type for each of the containers abstracted within.

Workflow and UML design: After completing the student class which was required for two other main driver classes. We split the functions of the other three classes before coming back together to test the code. UML was designed to be as simple as possible. We used names and variables that were clear to the reader what each function or variable stood for.

We used a testing strategy principle of testing individual parts of the program while adhering to our workflow strategy. As the program was divided into parts, each part had it's individual planning, designing, and testing. Using this approach, we were able to test each class on their own to make sure they worked before using them with other parts of the program.

¹ Abstraction: Abstraction is a method of encapsulation, hiding the inner working and inner components, limiting the visibility to the scope defined by the programmer. (Glossary pg. 16).

SDLC Flow and plan:



[i]

The *Software Development Life Cycle (SDLC)*² follows the sequence shown above. Starting at the planning stage of the software development life cycle, the programming team takes the *specification*³ 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*⁴ 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

² Software Development Life Cycle: Software Development Life Cycle: The Software Development Life Cycle may differ depending on the model and the application at play, however generally the Life Cycle consists of a planning phase, an analysis phase, a design phase, an implementation phase, and a testing phase. The Software Development Life Cycle may be abbreviated as SDLC. (Glossary pg. 18)

³ Specification: The specification of a project is the task which has been assigned to the programming team. The specification should also include the scope of work and the expectations of the job. Generally included in the specification is the medium in which the work is to be submitted and the desired format as well. (Glossary pg. 18)

⁴ Analysis and Design: Analysis and Design is the second stage of the SDLC, in which the specification is broken down and UML and Flow charts are started and completed as the analysis and design progress. It is important to consider the different factors such as compile time, memory allocation and efficiency when analyzing and designing the program based on the given specifications. The time analysis can be done using Big-O notation. (Glossary pg. 18)

efforts to optimize them. The different *implementations*⁵ and components⁶ of the program are mapped out, in forms such as *hierarchy charts*⁷, *pseudo code*⁸, *flow charts*⁹ and more *logical design elements*¹⁰. 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*¹¹.

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.

⁵ Implementation: Implementation: Implementation is the fourth stage of the SDLC. During this stage, the code starts to come together as the code is translated from the logical designs such as UMLs, into source code. (Glossary pg. 17)

⁶ Components: Components are the inner working parts of a whole. When applying abstraction, components are the inner working parts which are abstracted away from the end-user. (Glossary pg. 17)

⁷ Hierarchy Chart: A Hierarchy chart is a Logical Design Chart which is used to show the breakdown of the various functions and what tasks or methods they may branch off into. This is a helpful design element following the creation of the flow chart. Paired with the flow chart, one may get a better picture at the design, architecture and flow of the code and program. (Glossary Pg. 17).

⁸ Pseudo Code: Pseudo code is code that is abstracted at a level closer to human readable languages such as English. Pseudo code combines a coding language and another language in order to produce a rough outline of the functions and classes that are encapsulated within the Pseudo Code. (Glossary pg. 18).

⁹ Flow Chart: A Flow Chart is a graphical logical design element which describes the runtime flow of the program and provides a visual representation of how the code may operate. Another type of Flow Chart may be a Steak Diagram, which outlines the steps and conditions in a bubble format. (Glossary pg. 18).

¹⁰ Logical Design Forms: Logical Designs are those that help describe the logical process that goes into the design of the code, as well as the methods of operation. Logical Designs are crucial to the design phase of the Software Development Life Cycle. (Glossary pg. 18).

¹¹ Logic Errors: Errors are errors which occur in the logical flow of the program. They do not necessarily throw exceptions or cause the program to halt, however they do lead to unwanted results within the program and tend to be the hardest error to find. (glossary pg. 18).

Roles and Responsibilities:

Student Class:

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. One team member would write the relational operators, the other would write the input and output stream operators. After all the work in the class has been divided and completed, both team members come together once again in order to begin fully exercising the code. Starting by both team members reviewing the code in order to provide each other an extra reviewer prior to passing the class back to the main branch for the other team members to begin implementing the code in their sections. If one of the programmers assigned to the section has a problem or are stuck, they then would first ask the other programmer for help with debugging the issue if the issue persists. If the issue still persists, the error is then passed further up along the workflow stream, and the issue is brought to the group as a whole in the efforts to help remediate. Following finalizing the section, the code is then reviewed by another member of the group prior to the working branch being merged back with the main branch. If the issue persists beyond the first commit, an issue is raised in *GitHub*¹² in order to be resolved at a later time and to have proper documentation of the found issue.

VectorDriver Class

The vector is created in private so that team members can create objects of the student class. The accessors are created by a team member so that they can access the data in private, and the mutators were split between different team members to modify the data.

ListDriver Class

The list class creates a list and uses functions from the C++ STL in order to manipulate the list. Both team members oversee the framework of the source code. The lead role will create the accessors and mutators functions, and the other will have the responsibility to implement the functions.

IntContainer Class

The application section was divided up fairly simply since there are only four public methods and a driver. One person was responsible for working on the driver function and menu for the application section, while two people began working on the IntContainer Class. Following this division of work, the Application header file was able to be completed in a short amount of time with minimal errors and multiple people reviewing the code along the workflow.

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*¹³ 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*

¹² GitHub: Github is a free online resource which allows for teams to coordinate using the git protocol which is a workflow tracking protocol which keeps track of changes in the working directories. (Glossary pg. 18).

¹³ Agile Workflow: The agile workflow is a workflow process in which the tasks are divided up and completed asynchronously, reconvening at a regular frequency to redivide the work as time progresses. This allows for various parts of the code to be reviewed at asynchronous times and may allow for more review of code depending on the implementation of the workflow. The Agile workflow is more adaptive than the waterfall counterpart which is generally used in bigger teams and organizations. (Glossary pg. 19).

*Template Library (STL)*¹⁴, we were able to better implement the methods we designed that were efficient and logical for the Application class section.

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. The IntContainer Class was designed separately after all other parts. As there is no useful overlap in function. Thus, the IntContainer Class made no use of any other header file except for the input header file.

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 hold a List STL container using the Student Class as the underlying data type which the List is run upon. This required the student class to have various overloaded operators in order to better optimize the Student Class for use with the various STL functions and classes. The ListDriver adds a level of abstraction to the List STL container held within it, requiring the manipulation of the class to be done via the methods of the class, since the List is placed within a private visibility¹⁵. The ListDriver is designed with methods which have names which mimic and represent the functions run and called within each method. For example, the *pop_back* methods of the ListDriver runs the contained List's member function *pop_back()* on the List.

The VectorDriver class is designed to hold a vector STL container using the Student Class as the data type. Like the list class, this class also makes use of the overloaded operators within the student class. As the vector is private. The manipulation of the class is done by the functions within the public space. As with the ListDriver class, the functions of the vector class are named by the uses of those functions. For example, void *frontOfVector()* would return the first element of the vector and void *backOfVector* would return the last element in the vector.

The IntContainer class is a vector list designed to hold and keep track of numbers of integers. The container for the class is a list holding vectors and the vectors holding the amount of each integer. This container is private and thus requires the public functions to manipulate the value. The four public

¹⁴ Standard Template Library (STL): A general library of data structures and functions for lists, stacks and many more. (Glossary Pg. 19).

¹⁵ Visibility: The practice of standardizing the format and the style of a program code to a state of great consistency (Glossary pg. 19).

functions serve to add an integer, remove an integer, display the list of integers or show how frequent each integer occurs.

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.

Flow-Chart/ Program Hierarchy:

[See the Glossary for reference to the Flow-Chart and Program Hierarchy Chart.]

The Flow-Charts referenced below in the glossary were designed during the design phase of the SDLC. They provide a way of developing a deeper understanding of the flow of the code and allow for different analysis to be posed in regards to things such as time analysis, and efficiency along with the general program flow. The first Figure is a flow chart showing that the Student Class is the underlying data type of the VectorDriver and ListDriver classes while the application section IntContainer class is run on a list inside a vector. We decided to put a vector inside a list because a vector is random access and would make more logical sense for a container of a more fixed size such as the one used in the IntContianer. The list is useful in this way since it follows linear time and every value is passed over as it traverses the list, which improves efficiency when used as a container that will be displayed and or sorted more efficiently.

(Program Flow Chart : pg.14, Figure 1)

The next design elements referenced are both Hierarchy Charts showing the program broken into smaller components. This helped in the dividing up of the various tasks and how the program worked together with the other classes.

(Program Hierarchy : pg. 17, Figure 2)

(Program Hierarchy Zoomed In : pg. 16 , Figure 3)

UMLs:

[See the Glossary for reference to the UMLs.]

UMLs are design elements of classes outlining the methods and the components in the function along with description for each function and the preconditions for those functions. The UMLs describe the purpose of each function and act as a roadmap for the programmer as well as those who desire to implement the code.

(Student Class: pg. 17, Figure 4)

(VectorDriver Class: pg. 18, Figure 5)

(ListDriver Class: pg. 19, Figure 6)

(IntContainer Class: pg. 20, Figure 7)

Solution Details:

One of the main things we considered when creating our solution was the naming conventions used in order to improve reusability. Another thing taken in consideration when creating our solution was the implementation and its seamless integration with GitHub. When integrating with GitHub and using git, we were able to split up and simultaneously code as previously mentioned. This greatly improved the efficiency of our design, implementation and testing, all while providing everyone access to edit and view the code.

Main Components:

The program consists of driver functions which drive the classes

Classes:

Student
List
Vector
IntContainer

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:

The ListDriver has about thirty-six members, with only two private members and no private member methods. The private member *mListOne* is a list container that stores student objects and is the container in which the ListDriver provides a level of abstraction for. The next private member variable within the ListDriver is the *mDEBUG* constant boolean. *mDebug* acts as a switch turning on a more verbose response for debugging. This member constant is turned off or removed during normalization of the code. The constructor *ListDriver* will be called to initialize a list to be manipulated.

The ListDriver has about twenty-five mutator methods. The first mutator method in the ListDriver is the *clear()* method. It takes no parameters, and clears the list if the list is not empty. The method invokes the STL List Class method of the same name, *clear()*. The next mutator is the *resize()* method. The

resize() method doesn't take any parameters, however it utilizes the input.h input validation and sanitization, querying the end-user for a value to resize the list.

In the ListDriver there exists two methods of reading from a file. Both of these methods are mutator methods of the ListDriver Class. One method, *readFrontNPopulate()*, reads from an input file, named input.dat, case sensitive, contained in the same folder as the running application. It parses the file and populates Student objects and populates them into the list from the front. The other method, *readBackNPopulate()* reads the file of the same name, input.dat, and populates Student objects, placing them into the list from the back.

The next mutator method is *pop_front()*, which removes the front element of the list, and similarly *pop_back()* removes the last element of the list, the element shown by the *front()* method. The *front()* is an overloaded method, where the other signature of the method is a constant method providing constant access to the return value of *front*. The *front()* function returns a Student reference ¹⁶variable to the first value in the list. The *showFront()* method displays this reference variable to the standard output stream. The *showFront()* method is also overloaded in the same fashion as the *front()* method. Similarly to *front()*, the method *back()* is an overloaded mutator function with the other signature being a constant method providing access to the method in other constant methods and protecting the integrity of the list. The *back()* method, additionally like *front*, provides a Student reference variable to the last element in the list. Once again, similar to *front()*, *back()* is displayed to the standard output stream through the *showBack()* method.

This same format, of having a standard output stream display method for the various reference variables and iterators from the *mListOne* List held within the class, is applied with methods such as *showBegin()*, *showEnd()*, *showRbegin()*, and *showRend()*. Where each of these methods displays to the standard output stream the value held by the List Class methods following the prefix 'show'. Similarly to the List Class method called within the methods of the ListDriver class above, the underlying method is also abstracted in methods named the same as the method abstracted within it.

The method *begin()* returns the iterator pointing to the beginning of the *mListOne*, and similarly, *rbegin()* returns the reverse iterator pointing to the last value of the list. The *rend()* method returns the reverse iterator pointing to the past-the-end reverse iterator pointing to a theoretical value before the first element in the list. The *end()* method, similar to the *rend()* method, returns an iterator pointing to the past-the-end theoretical value. These values cannot be dereferenced, they only stand as sentinel values to be used to know when the iterator has reached the end or the beginning of the list.

The ListDriver Class also has an *emptyCheck()* method which checks if the *mListOne* is empty and if it is it prints out to the standard output that the list is empty, and then it returns the boolean representing the empty state of the list. This method is utilized in both constant and mutator methods.

In addition to the methods returning attributes of the *mListOne*, there is an *erase()* method, overloaded with one and two parameters to match those of the overloaded functions abstracted within. These methods will remove the value at the iterator, or remove a range of values between two iterators. Additionally, the *insert()* method will insert a value at the one argument passed to it, an iterator for the position to insert the new element. The swap method will swap two STL List Class objects running on the Student Class as the underlying data type.

¹⁶ Reference Variable: A reference variable is a variable that passes the memory address and references the original data, so when the variable is manipulated, the original variable is manipulated, versus the alternative which passes a copy of the variable, which in terms of parameters is known as a pass by value versus pass by reference. (Glossary pg. 19).

VectorDriver:

The Vector Driver has about forty-five members. Only one vector container is private. Many of the members only serve to obtain a vector as an argument to access a different public function to generate the result. This allows the use of a different function to check for certain conditions before running the function that would create the desired effect on the vector.

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 by the *vectorIsEmptyMethod* to see if the vector is empty. Which would return false if the vector is empty or would return true and output a statement if it is empty.

The *clearMethod* mutator would take in a vector argument and pass it to *clearVector*. This function would run the STL *vector::clear* function for the argument vector. The *reserveMethod* mutator takes in a vector argument and passes it to *reserveVector*. This function would ask for a user input integer number from 1-100 and reserve space for the vector using STL *vector::reserve* function. The *resizeMethod* function works the same as the *reserveMethod*. It takes a vector argument and passes it to *resizeVector*. The function would then ask for an integer number from 1-100 and change the size of the vector using STL *vector::resize* function.

Next, *readFileMethod* would take the augmented vector. Create a temp student class and open up input.dat in read mode. Then, it reads the file and streams it into the temp student class. The temp student class would be passed on to the *pushBackToVector* function. This function would run STL *vector::push_back* and add this on to the end of *mMyVector*. This would repeat until the file ends.

The *popBackMethod* would take the vector and use *vectorIsEmptyMethod* to check if the vector is empty before passing it to *popBackVector*. This function uses STL *vector::pop_back* to delete the last element in the vector.

The three functions *frontMethod*, *backMethod* and *indexMethod* all display an element without using iterators. First they all check if the vector is empty using *vectorIsEmptyMethod*. Next, *frontMethod* calls *frontOfVector*, *backMethod* calls *backOfVector* and *indexMethod* gets an integer from the user before calling and passing the integer to *indexOfVector*. The *frontOfVector* displays the first element in the vector using STL *vector::front*. The *backOfVector* displays the last element in the vector using STL *vector::back*. The *indexOfVector* displays the nth element in the vector using STL *vector::at*. The n is based on the integer from the user input.

Next two functions are *iterBegin* and *iterEnd*. They both first check if the vector is empty. After that, *iterBegin* calls *iBegin* and *iterEnd* calls *iEnd*. *iBegin* creates a vector iterator and sets it to the first element in the vector using STL *vector::begin*. *iEnd* creates a vector iterator and sets it to the memory address after the last element in the vector using STL *vector::end*. They then both display their address and what is in it.

The *iterReturn* function returns all elements in the vector using iterators. After checking if the list was empty, it calls *iReturn*. This function uses a for statement and sets the iterator to the beginning of the vector move one by one through all the elements in the vector. It ends when the iterator is equal to the address after the last element in the vector. It will display the vector element from the first element to the last element.

iterRBegin and *iterREnd* are similar to *iterBegin* and *iterEnd*. They check for an empty list. *iterRBegin* calls *iRBegin* which uses STL *vector::rbegin* to set a reverse iterator to the last element of the list and displays it. *iterREnd* calls *iREnd* and uses STL *vector::rend* to set a reverse iterator to the memory address in front of the first element in the vector.

iterRReturn calls *vectorIsEmptyMethod*. If it is false, then it calls *iRReturn*. This function uses a for statement and sets it to start with the reverse iterator at the last element. Then it prints out every element until the reverse iterator is pointing at the memory address in front of the first element in the vector.

iterErase calls *eraseFromVector* after checking if the vector is empty. *eraseFromVector* set an iterator on the first and delete it with STL *vector::erase*.

eraseVectorRangeMethod calls *eraseRangeVector* if the vector is not empty. This would set one iterator with STL *vector::begin* and a second iterator with STL *vector::end*. Then these two iterators would be passed to STL *vector::erase* to delete all elements of the vector.

iterInsert calls *insertEntry* after verifying there are elements in the vector. It first creates a newStudent student class and an iterator at the first element of the vector. After setting up the student with the information. it adds the newStudent to the front of the vector using STL *vector::insert*.

swapMethod runs *swapVector* after checking there are elements in the vector. *swapVector* will create a new vector container called v2. Next it uses STL *vector::swap* to swap the content of mMyVector to v2 and then display all the elements of v2.

iterSort runs *sortVector* after checking for elements in the vector. *sortVector* sets one iterator to STL *vector::begin* and a second one to STL *vector::end*. Then it passes the two iterators to STL *vector::sort* to sort the vector.

vectorMenuOption displays a menu and obtains a char input from the user to choose which function method to run.

IntContainer:

The private member *mListVector* is a list object that contains vectors as elements. IntContainer calls an empty default constructor since the mListVector is instantiated with the list<vector<int>> default constructor. The *addNumber()* method is a void function that will prompt the user to add an integer value. Once an integer value is inputted then the program will check if the vector is empty. If *mListVector* is empty then it declares a vector name *freq*. The value inputted is pushed into *freq* and then the value of one, then the vector with the two elements is pushed into *mListVector*. If the *mListVector* is not empty then the function will iterate through each list element comparing the first element of the vectors and determine if it's equal to the value provided by the user. If the value is present then it will increment the second element of that vector. If in that iteration there's no value found that matches, then a new vector named *freq* is declared and the value given is pushed to the first element and then the number one is pushed after. This function will keep a list with vectors of two elements. Every vector would hold in the first element the number inputted by the user and the second element will represent the number of times that number has been added.

removeNumber() is a function that starts by initiating a boolean name found to false. It prompts the user an integer to delete. The function then it will iterate through the will iterate through the each list element comparing the first element of the vectors and determine if it's equal to the value provided by the user. If the number does equal to one of the values in the first elements in the vectors then it will decrement the

size of the second element associated with that of the value given. If the second element in the vector decremented equals zero or less, it would be erased from the *mListVector*. If the number provided by the user is not found then the function will prompt a message that would state that the value provided is not found.

displaySet() is a void function that will iterate through all the first elements in the vectors and output the values. *displayFrequencies()* is a void function that will iterate through all the two elements in the vectors and output the values displaying the values inputted and their frequencies.

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*¹⁷ 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*¹⁸ 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 IntelliTips¹⁹ documentation within visual studio to better allow reading of documentation in Microsoft Visual Studio when implementing

¹⁷ Normalization: The practice of standardizing the format and the style of a program code to a state of great consistency. (Glossary pg. 19).

¹⁸ Camel Hump Format: The way of writing out expressions in which there are no spaces or underscores and the first letter of the first word is written in lowercase and the following words are started with an upper case letter (e.g. exampleOfCamelCase). (glossary pg. 19).

¹⁹ IntelliTips: IntelliTips are small toolkit style popup boxes which appear when typing in visual studio. They provide tips and recommendations as well as live documentation based on an internal XML document which Visual Studio maintains. (Glossary pg. 19).

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”²⁰ 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 its strict weak order.

²⁰ <https://www.cplusplus.com/reference/list/list/sort/>

References

Agile Alliance. *Agile 101*. n.d. Website. October 2021. <<https://www.agilealliance.org/agile101/>>.

Altexsoft. *Technical Documentation in Software Development Types Best Practices and Tools*. 01 December 2020. Website. October 2021. <<https://www.altexsoft.com/blog/business/technical-documentation-in-software-development-types-best-practices-and-tools/>>.

C++ Reference Guides. *Sort*. n.d. Website. October 2021. <<https://www.cplusplus.com/reference/list/list/sort/>>.

Ivanna. *types-of-software-development-documentation*. Ed. Alexandra Rostovtseva. 22 May 2020. Website. October 2021. <<https://gbksoft.com/blog/types-of-software-development-documentation/>>.

SCRUM.org. *What is SCRUM*. n.d. Website. October 2021. <<https://www.scrum.org/resources/what-is-scrum/>>.

geeksforgeeks.org *The C++ Standard Template Library (STL)* 6/28/2021. Website. October 2021. <<https://www.geeksforgeeks.org/the-c-standard-template-library-stl/>>

Gaddis, Tony. "Introduction to C++." Starting Out with C from Control Structures to Objects, 9th ed., Pearson Education, Inc, 2017 , print.

Glossary:

Terms:

1. **Abstraction:** Abstraction is a method of encapsulation, hiding the inner working and inner components, limiting the visibility to the scope defined by the programmer.
2. **Software Development Life Cycle:** The Software Development Life Cycle may differ depending on the model and the application at play, however generally the Life Cycle consists of a planning phase, an analysis phase, a design phase, an implementation phase, and a testing phase. The Software Development Life Cycle may be abbreviated as SDLC.
3. **Specification:** The specification of a project is the task which has been assigned to the programming team. The specification should also include the scope of work and the expectations of the job. Generally included in the specification is the medium in which the work is to be submitted and the desired format as well.
4. **Analysis and Design:** Analysis and Design is the second stage of the SDLC, in which the specification is broken down and UML and Flow charts are started and completed as the analysis and design progress. It is important to consider the different factors such as compile time, memory allocation and efficiency when analyzing and designing the program based on the given specifications. The time analysis can be done using Big-O notation.
5. **Implementation:** Implementation is the fourth stage of the SDLC. During this stage, the code starts to come together as the code is translated from the logical designs such as UMLs, into source code.
6. **Components:** Components are the inner working parts of a whole. When applying abstraction, components are the inner working parts which are abstracted away from the end-user.
7. **Hierarchy Chart:** A Hierarchy chart is a Logical Design Chart which is used to show the breakdown of the various functions and what tasks or methods they may branch off into. This is a helpful design element following the creation of the flow chart. Paired with the flow chart, one may get a better picture at the design, architecture and flow of the code and program.
8. **Pseudo Code:** Pseudo code is code that is abstracted at a level closer to human readable languages such as English. Pseudo code combines a coding language and another language in order to produce a rough outline of the functions and classes that are encapsulated within the Pseudo-Code.
9. **Flow Chart:** A Flow Chart is a graphical logical design element which describes the runtime flow of the program and provides a visual representation of how the code may operate. Another type of Flow Chart may be a Steak Diagram, which outlines the steps and conditions in a bubble format.
10. **Logical Design Forms:** Logical Designs are those that help describe the logical process that goes into the design of the code, as well as the methods of operation. Logical Designs are crucial to the design phase of the Software Development Life Cycle.
11. **Logic Errors:** Logic Errors are errors which occur in the logical flow of the program. They do not necessarily throw exceptions or cause the program to halt, however they do lead to unwanted results within the program and tend to be the hardest error to find.
12. **GitHub:** Github is a free online resource which allows for teams to coordinate using the git protocol which is a workflow tracking protocol which keeps track of changes in the working directories.
13. **Agile Workflow:** The agile workflow is a workflow process in which the tasks are divided up and completed asynchronously, reconvening at a regular frequency to redivide the work as time progresses. This allows for various parts of the code to be reviewed at asynchronous times and may allow for more review of code depending on the implementation of the workflow. The Agile

workflow is more adaptive than the waterfall counterpart which is generally used in bigger teams and organizations.

14. Standard Template Library (STL): A general library of data structures and functions for lists, stacks and many more.
15. Normalization: The practice of standardizing the format and the style of a program code to a state of great consistency.
16. Reference Variable: A reference variable is a variable that passes the memory address and references the original data, so when the variable is manipulated, the original variable is manipulated, versus the alternative which passes a copy.
17. Camel Hump Format: The way of writing out expressions in which there are no spaces or underscores and the first letter of the first word is written in lowercase and the following words are started with an upper case letter (e.g. `exampleOfCamelCase`)
18. IntelliTips: IntelliTips are small toolkit style popup boxes which appear when typing in Visual Studio. They provide tips and recommendations as well as live documentation based on an internal XML document which Visual Studio maintains.

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 define 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

Conditional Documentation:

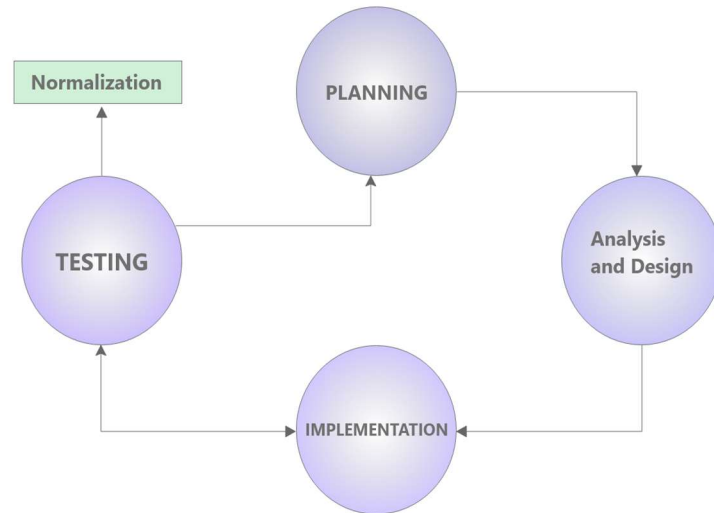
Prior to functions, the documentation of the precondition and postcondition are described with three brackets so that the descriptions would work with Microsoft Visual Studio IntelliSense, which produces would then display the precondition and postcondition when as a tip when the function is being implemented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

Class Invariants:

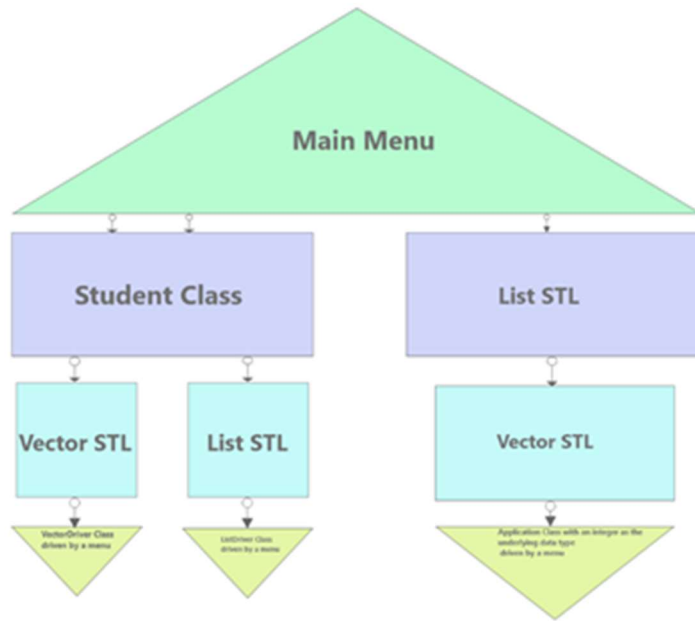
Class invariants provide a list of the various methods in each class and provide a short description outlining the use and implementation notes for that class. The methods in the class invariant don't provide formal parameters but instead they show the datatype of each formal parameter in a similar format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

Flow Charts:

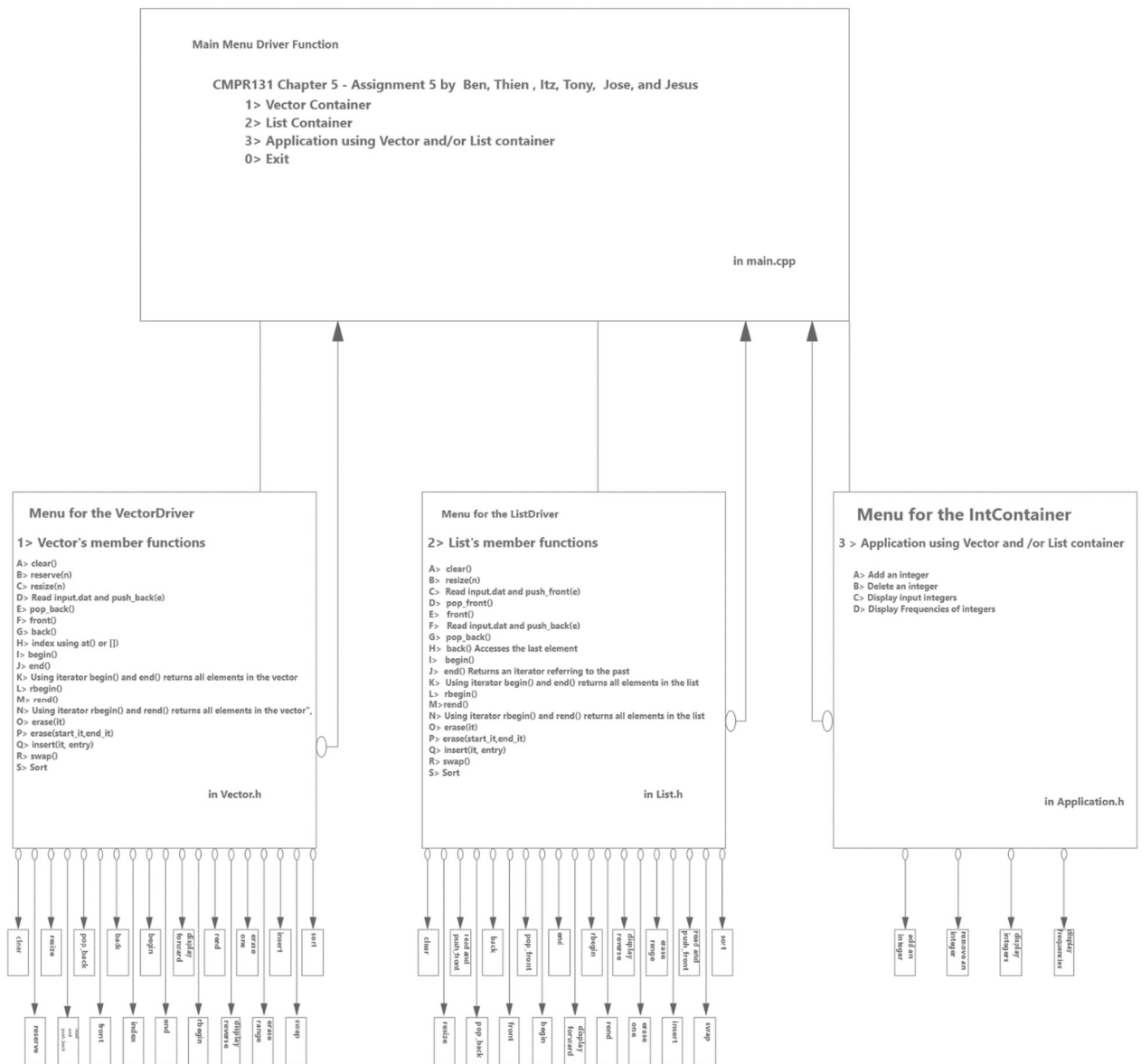


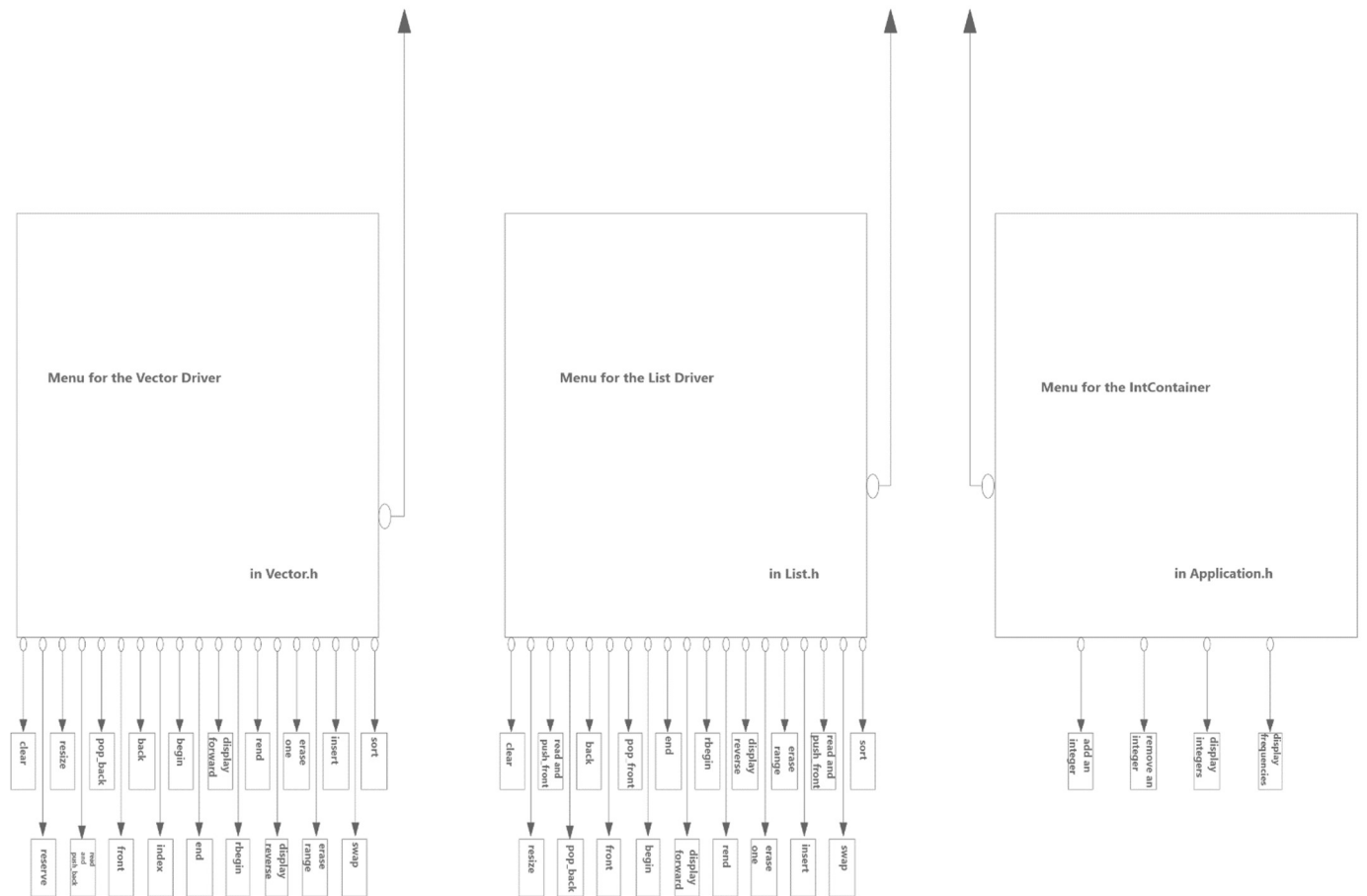
[i]



[ii]

Hierarchy Charts:





[iv]

UMLs

[v]

Student Class UML	
- mName : int const - mLEVELS_ARRSIZE : int const - mLEVELS[] : string const - mLevel : int - mGpa : double - mEmpty : bool - mError : bool - mDEBUG : bool const - mCheckEmpty() : void	(String) the name of the student Holds the constant array size Constant array of strings for levels of student Holds the integer value for the student level holds the students gpa true if not initialized and empty true if an error occurs debug switch Precondition: N/A Postcondition: checks if the empty
Constructors	Constructors
+ Student(pName : string, pNLevel : int, pGpa : double)	Precondition: N/A Postcondition: declaring an object for the student, initializes the string, int and double to default values if no parameters given
+ student(copy : Student& const)	Precondition: copy is a copy of the student object to be copied into this Postcondition: Copy constructor, copies one student to the next
Accessors	Accessors
+ getName() : string const	Precondition: N/A Postcondition: returns the name value of the student
+ getLevel() : string const	Precondition: N/A Postcondition: returns the string equivalent to the student level
+ getNLevel() : int const	Precondition: N/A Postcondition: returns the integer level of the student
+ getGpa() : double const	Precondition: N/A Postcondition: returns the gpa of the student
+ empty() : bool const	Precondition: N/A Postcondition: returns true if the student object is empty
+ error() : bool const	Precondition: N/A Postcondition: returns true if the error bit is on
+ operator == (obj : student& const) : bool const	Precondition: (student&) obj is the student object to the right Postcondition: returns true if the gpa, strLevel, and name are all equal
+ operator <= (obj : student& const) : bool const	Precondition: (student&) obj is the student object to the right Postcondition: returns true if the gpa, strLevel, and name are all less than or equal to the object
+ operator >= (obj : student& const) : bool const	Precondition: (student&) obj is the student object to the right Postcondition: comparing objects of two students by the less than or equal sign
+ operator < (obj : student& const) : bool const	Precondition: (student&) obj is the student object to the right of the comparison Postcondition: returns true if the gpa or the name are less than the object
+ operator > (obj : student& const) : bool const	Precondition: (student&) obj is the student object to the right Postcondition: returns true if the gpa or the name are greater than the object
Mutators	Mutators
+ setName() : void	Precondition: N/A Postcondition: prompts the user through a sanitized input and then sets that to the gpa
+ setName(pName : string) : void	Precondition: N/A Postcondition: prompts the user through a sanitized input and then sets that to the gpa
+ setLevel() : void	Precondition: N/A Postcondition: prompts the user through a sanitized input and then sets that to the level
+ setLevel(pLevel : int) : void	Precondition: (int) pLevel is an value 0-3 representing the index of the level Postcondition: prompts the user through a sanitized input and then sets that to the level
+ setLevel(pLevel : string) : void	Precondition: (string) pLevel is the level as a string value, case sensitive Postcondition: prompts the user through a sanitized input and then sets that to the level
+ setGpa() : void	Precondition: N/A Postcondition: prompts the user through a sanitized input and then sets that to the gpa
+ setGpa(pGpa : double) : void	Precondition: N/A Postcondition: prompts the user through a sanitized input and then sets that to the gpa
+ empty() : bool	Precondition: N/A Postcondition: returns true if the student object is empty
+ error() : bool	Precondition: N/A Postcondition: returns true if the student object is empty
+ operator ==(obj : student& const) : bool	Precondition: (Student&) obj is the student object to the right Postcondition: returns true if the gpa, strLevel and name are all equal to that of the object
+ operator <=(obj : student&) : bool	Precondition: (Student&) obj is the student object to the right Postcondition: returns true if the gpa, strLevel and name are all less than or equal to the object
+ operator >=(obj : student&) : bool	Precondition: (Student&) obj is the student object to the right Postcondition: comparing objects of two students by the less than or equal sign
+ operator = (obj : student& const) : void	Precondition: (const Student&) obj is an object of the Student class to the right of the assignment, being assigned to the object Postcondition: Copies the private members of the (const student&) obj to the Student object called from.
+ operator = (obj : student&) : void	Precondition: (Student&) obj is an object of the Student class to the right of the assignment, being assigned to the object Postcondition: Assigns the private members of the (const student&) obj to the Student object called from.
+ operator < (obj : student&) : bool	Precondition: Precondition: (Student&) obj is the student object to the right of the comparison Postcondition: Postcondition: returns true if the gpa or the name are less than the object
+ operator > (obj : student&) : bool	Precondition: (Student&) obj is the student object to the right Postcondition: returns true if the gpa or the name are greater than the object
Friend Functions	
+ operator <<(strm : ostream& strm, obj : student& const) : ostream& friend	Precondition: output stream used with an output stream object and the output stream operator << Postcondition: displays to the stream the student in the format (name, level, gpa)
+ operator <<(strm : ostream& strm, obj : student&) : ostream& friend	Precondition: output stream used with an output stream object and the output stream operator << Postcondition: displays to the stream the student in the format (name, level, gpa)
+ operator >>(strm : istream& strm, obj : student&) : istream& friend	Precondition: Postcondition: populates the student from the standard input stream(name, level, gpa)
+ operator >>(strm : fstream&, obj : student&) : fstream& friend	Precondition: Postcondition: populates the student form the standard file stream (name, level, gpa)

ListDriver Class UML	
Private	
- mListOne : list<student>	student class list for holding student information
- mDEBUG : bool const	debug switch
Constructor	
+ ListDriver()	Creates an empty list
Accessors	
+ emptyCheck() const : bool	Precondition: list initiated Postcondition: checks if list is empty
+ displayAll_reverse() const : void	Precondition: list initiated Postcondition: displays all elements in reverse
+ showFront() const: void	Precondition: list is initialized and not empty Postcondition: returns the first element of the list in a text format
+ showBack() const: void	Precondition: list cannot be empty Postcondition: returns the last element of the list in a text format
+ showRbegin() const: void	Precondition: list cannot be empty Postcondition: returns the memory location of the rbegin in a text format
+ showRend() const: void	Precondition: list cannot be empty Postcondition: returns the memory location of the rend in a text format
+ showBegin() const: void	Precondition: list cannot be empty Postcondition: returns the memory location of the begin in a text format
+ showEnd() const : void	Precondition: N/A Postcondition: returns the address of the end()
Mutators	
+ clear() : void	Precondition: list is initiated Postcondition: list is cleared
+ resize() : void	Precondition: (int) n must be greater than 0 and less than 100 Postcondition: resizes the allocated amount in the list mListOne
+ readFrontNPopulate() : void	Precondition: input.dat in project folder Postcondition: read file using push_front
+ pop_front() : void	Precondition: list initiated Postcondition: front element removed
+ showFront() : void	Precondition: list cannot be empty Postcondition: returns the first element of the list in a text format
+ readBackNPopulate() : void	Precondition: input.dat in project folder Postcondition: read file using push_back
+ pop_back() : void	Precondition: list cannot be empty Postcondition: back element removed
+ showBack() : void	Precondition: list cannot be empty Postcondition: returns the last element of the list in a text format
+ showBegin() : void	Precondition: list cannot be empty Postcondition: returns the memory location of the begin in a text format
+ showEnd() : void	Precondition: list cannot be empty Postcondition: returns the memory location of the end in a text format
+ displayAll() : void	Precondition: list cannot be empty Postcondition: displays all elements
+ showRBegin() : void	Precondition: list cannot be empty Postcondition: returns the memory location of the rbegin in a text format
+ showRend() : void	Precondition: list cannot be empty Postcondition: returns the memory location of the rend in a text format
+ begin() : list<student>::iterator	Precondition: list initiated Postcondition: creates an iterator and outputs beginning element and iterator
+ end() : list<student>::iterator	Precondition: list initiated Postcondition: creates an iterator and outputs last elements iterator
+ erase(pltt : list<Student>::iterator) : void	Precondition: it is an iterator to the value to be remove Postcondition: removes value at position
+ erase(pStart : list<Student>::const_iterator, pEnd : list<student>::const_iterator): list<student>::iterator	Precondition: pStart iterator is the start of the value to be deleted, end is the end iterator to be deleted exclusive Postcondition: removes the range
+ insert() : list<Student>::iterator	Precondition: N/A Postcondition: inserts a student object in the list
+ swap() : void	Precondition: N/A Postcondition: swaps the list with an empty list (clears the list indirectly)
+ sort() : void	Precondition: list cannot be empty Postcondition: list is sorted alphabetically
+ emptyCheck() : bool	Precondition: list initiated Postcondition: checks if list is empty

[vi]

VectorDriver class UML	
- myVector: vector	Contains a list of students
Constructor	
+ vectorDriver()	Precondition: N/A Postcondition: initializes default vector
Accessors	
+ getSizeOfVector() : int	Precondition: VectorDriver object must be initialized Postcondition: will return the number of students in the myVetorDriver object
Mutators	
+ clearVector() : void	Precondition: VectorDriver object must be initialized Postcondition: clears all the elements from the vectorDriver
+ reserveVector() : void	Precondition: VectorDriver object must be initialized Postcondition: will reserve the space for the given value for the user
+ resizeVector() : void	Precondition: VectorDriver object must be initialized Postcondition: will resize the vector by the value given by the user
+ popBackVector() : void	Precondition: there must be elements in the object Postcondition: will return the last value inputted
+ frontOfVector() : void	Precondition: there must be elements in the object Postcondition: will return the first element (positioned in the front)
+ backOfVector() : void	Precondition: there must be elements in the object Postcondition: will return the last element added
+ pushBackToVector(pValue: Student) : void	Precondition: the object must be initialized and the value must be a student object Postcondition: will add the student object to the end of the vector
	Precondition: there must be a student object in the given index Postcondition: will return the student in the given object
+ indexOfVector(pIndex: int) : void	
	Precondition: there must be students in the vectorDriver object Postcondition: will output the memory address and the value of the first element using iterator
+ iBegin() : void	Precondition: there must be students in the vectorDriver object Postcondition: will output the memory address and the value of the last element using iterator
+ iEnd() : void	Precondition: there must be students in the vectorDriver object Postcondition: will output the memory address and the value of the elements from begin() to end() using iterator
+ iReturn() : void	Precondition: there must be students in the vectorDriver object Postcondition: will output the last element using reverse iterator
+ irBegin() : void	Precondition: there must be students in the vectorDriver object Postcondition: will output the last element using reverse iterator
+ irEnd() : void	Precondition: there must be students in the vectorDriver object Postcondition: will output the elements in reverse order using reverse iterator
+ irReturn() : void	
+ readFile() : void	Precondition: File must exist and the vectorDriver object must be initialized Postcondition: will push each student object onto the vectorDriver object using a CSV file
+ insertEntry() : void	Precondition: The VectorDriver object must be initialized Postcondition: Will insert a student into the vectorDriver object
+ swapVector() : void	Precondition: The VectorDriver object must be initialized Postcondition: will swap the values with another vector <student> that is initialized
+ sortVector() : void	Precondition: there must be students in the VectorDriver object Postcondition: will sort the vector in alphabetical order by iterator
+ eraseFromVector() : void	Precondition: there must be students in the VectorDriver object Postcondition: will erase the first element using iterator
+ eraseRangeVector() : vector<Student>::iterator	Precondition: there must be students in the VectorDriver object Postcondition: will erase elements by a range of iterators
+ vectorIsEmptyMethod(pMyVector : VectorDriver) : bool	Precondition: VectorDriver must be initialized Postcondition: will return true if there are no elements in the vectorDriver object or false if there is at least one student
+ eraseVectorRangeMethod(pMyVector : VectorDriver&) : void	Precondition: there must be students in the VectorDriver object Postcondition: will erase elements by a range of iterators
+ iterErase(pMyVector : VectorDriver&) : void	Precondition: VectorDriver must be initialized and it must have students in the object Postcondition: will erase the first element using iterator
+ indexMethod(pMyVector : VectorDriver) : void	Precondition: VectorDriver must be initialized and have students in the object Postcondition: will output the element in the VectorDriver given relative to the index given by the user
+ backMethod(pMyVector : VectorDriver) : void	Precondition: VectorDriver must be initialized and have students in the object Postcondition: will output the last element of the VectorDriver object
+ frontMethod(pMyVector : VectorDriver) : void	Precondition: VectorDriver must be initialized and have students in the object Postcondition: will output the first element of the VectorDriver object
+ popBackMethod(pMyVector : VectorDriver&) : void	Precondition: VectorDriver must be initialized and have students in the object Postcondition: will remove the last element of the VectorDriver object and output the elements of the object
+ readFileMethod(pMyVector : VectorDriver&) : void	Precondition: file must exists and must be a CSV format Postcondition: will input the student object fields and place the students in the VectorDriver object
+ resizeMethod(pMyVector : VectorDriver&) : void	Precondition: VectorDriver must be initialized Postcondition: will resize the VectorDriver object by the given size from the user
+ clearMethod(pMyVector : VectorDriver&) : void	Precondition: VectorDriver must be initialized Postcondition: will remove all the elements in the VectorDriver object
+ reserveMethod(pMyVector : VectorDriver&) : void	Precondition: VectorDriver must be initialized Postcondition: wil reserve the space for the VectorDriver object
+ iterBegin(pMyVector : VectorDriver&) : void	Precondition: VectorDriver must be initialized and must have elements Postcondition: will return the first element of the VectorDriver method
+ iterEnd(pMyVector : VectorDriver&) : void	Precondition: VectorDriver must be initialized and must have elements Postcondition: will return the last element of the VectorDriver method
+ iterReturn(pMyVector : VectorDriver&) : void	Precondition: there must be students in the VectorDriver object Postcondition: will output the elements in reverse order using reverse iterator
+ iterRBegin(pMyVector : VectorDriver&) : void	Precondition: there must be students in the VectorDriver object Postcondition: will output the last element using reverse iterator
+ iterREnd(pMyVector : VectorDriver&) : void	Precondition: there must be students in the VectorDriver object Postcondition: will output the last element using reverse iterator
+ iterRReturn(pMyVector : VectorDriver&) : void	Precondition: there must be students in the VectorDriver object Postcondition: will output the elements in reverse order using reverse iterator
+ iterInsert(pMyVector : VectorDriver&) : void	Precondition: The VectorDriver object must be initialized Postcondition: Will insert a student into the VectorDriver object
+ swapMethod(pMyVector : VectorDriver&) : void	Precondition: there must be students in the VectorDriver object Postcondition: will swap all the elements with an empty object
+ iterSort(pMyVector : VectorDriver&) : void	Precondition: there must be students in the VectorDriver object Postcondition: will sort and output the elements using iterator
Friend	
+ friend ostream& operator << (strm : ostream&, obj : VectorDriver& const) : ostream& friend	Precondition: N/A Postcondition: will output all of the elements in the vector

[vii]

IntContainer Class UML	
- mListVector: List << Vector: << Int >> >>	Contains a list of vectors of ints
Constructor	
+ IntContainer()	Postcondition: initializes default list
Mutators	
+ addNumber() : void	Precondition: N/A Postcondition: pushes an int inside a vector
+ remove() : void	Precondition: the container must not be empty Postcondition: deletes inputted value
Accessors	
+ displaySet() : void	Precondition: N/A Postcondition: If the container is not empty, displays values in container, displaying repeated values next to each other.
+ displayFrequencies() : void	Precondition: N/A Postcondition: displays frequencies of values inputted

[viii]

Table of Figures:

Figure 1 Software Development Life Cycle	3,14
Figure 2 Program Flow	14
Figure 3 Program Hierarchy	15
Figure 4 Zoomed in Hierarchy Chart	16
Figure 5 Student Class UML	17
Figure 6 VectorDriver UML	18
Figure 7 ListDriver UML	19
Figure 8 IntContainer UML	20

[i] Figure 1: Software Development Life Cycle

[ii] Figure 2: Program Flow Chart

[iii] Figure 3: Program Hierarchy Chart

[iv] Figure 4: Program Hierarchy Chart zoomed

[v] Figure 5: Student Class UML

[vi] Figure 6: VectorDriver Class UML

[vii] Figure 7: ListDriver Class UML

[viii] Figure 8: IntContainer Class UML