Winter Semester 2019

AID MANAGEMENT APPLICATION (AMA)

Version 3.6.1

[Update 19.03.2019]: added a query to class Product to retrieve the name of the product.

[Update 19.03.2019]: extended the deadline for the last milestone by a week.

When disaster hits a populated area, the most critical task is to provide immediately affected people with what they need as quickly and as efficiently as possible.

This project completes an application that manages the list of goods that need to be shipped to the disaster area. The client application tracks the quantity of items needed, tracks the quantity on hand, and stores the information in a file for future use.

The types of goods that need to be shipped are of two categories;

- Non-Perishable products, such as blankets and tents, which have no expiry date. We refer to products in this category as Product objects.
- Perishable products, such as food and medicine, that have an expiry date. We refer to products in this category as Perishable.

To complete this project you will need to create several classes that encapsulate your solution.

OVERVIEW OF THE CLASSES TO BE DEVELOPED

The classes used by the client application are:

Date

A class to be used to hold the expiry date of the perishable items.

ErrorState

A class to keep track of the error state of client code. Errors may occur during data entry and user interaction.

Product

A class for managing non-perishable products.

Perishable

A class for managing perishable products. This class inherits the structure of the "Product" class and manages an expiry date.

iProduct

An interface to the Product hierarchy. This interface exposes the features of the hierarchy available to the client application. Any "iProduct" class can

- read itself from or write itself to the console
- save itself to or load itself from a text file
- compare itself to a unique C-string identifier
- determine if it is greater than another product in the collating sequence
- report the total cost of the items on hand
- describe itself
- update the quantity of the items on hand
- report its quantity of the items on hand
- report the quantity of items needed
- accept a number of items

Using this class, the client application can

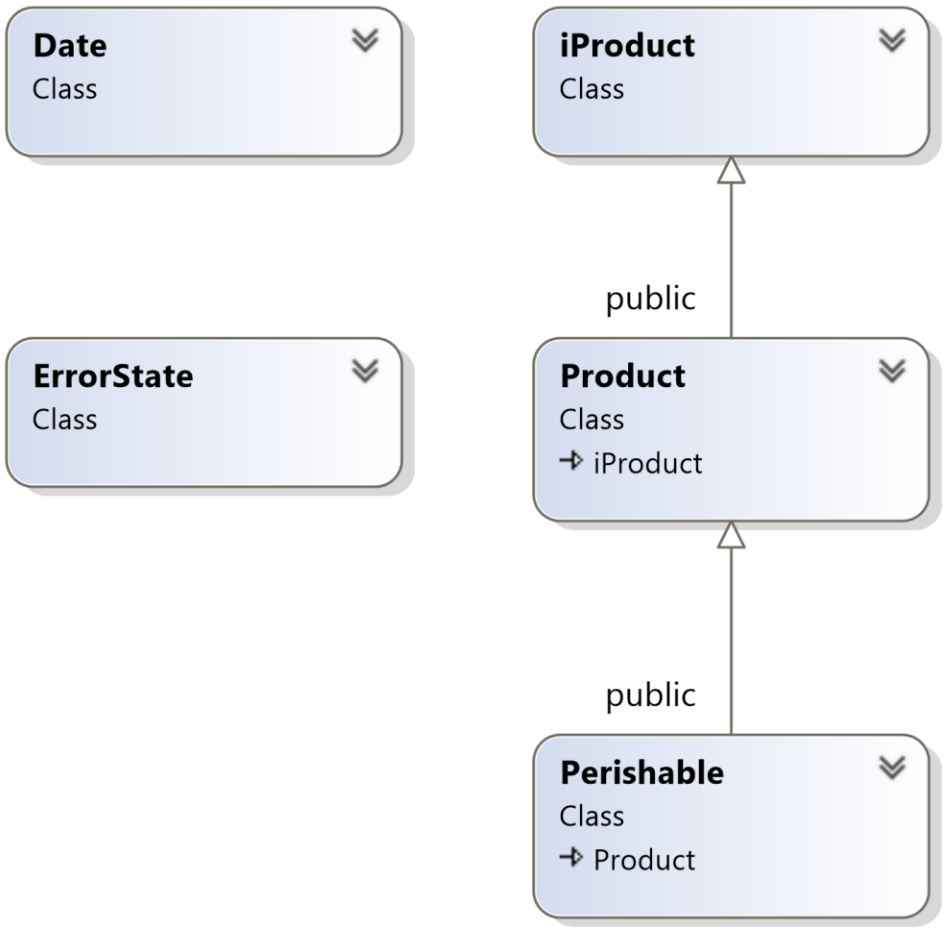
- save its set of iProducts to a file and retrieve that set later
- read individual item specifications from the keyboard and display them on the screen
- update information regarding the number of each product on hand

THE CLIENT APPLICATION

The client application manages the iProducts and provides the user with options to

- list the Products
- display details of a Product
- add a Product
- add items of a Product
- update the items of a Product
- delete a Product
- sort the set of Products

Project Class Diagram



Project Development Process

The Development process of the project consists of 5 milestones and therefore 5 deliverables. Shortly before the due date of each deliverable a tester program and a script will be provided for testing and submitting the deliverable. The approximate schedule for deliverables is as follows

- Due Dates
  - The `Date` class — Due: March 10th
  - The `ErrorState` class — Due: March 17th
  - The `Product` class — Due: March 24th
  - The `iProduct` interface — Due: March 27th
  - The `Perishable` class — Due: April 7th

General Project Submission

In order to earn credit for the whole project, you must complete all milestones and assemble them for the final submission. Incomplete projects receive 0%.

Note that at the end of the semester you **MUST submit a fully functional project to pass this subject**. If you fail to do so, you will fail the subject. If you do not complete the final milestone by the end of the semester and your total average, without your project's mark, is above 50%, your professor may record an "INC" (incomplete mark) for the subject. With the release of your transcript you will receive a new due date for completion of your project.

The maximum project mark that you will receive for completing the project after the original due date will be "**49%**" of the project mark allocated on the subject outline.

File Structure of the Project

Each class has its own header (.h) file and its own implementation (.cpp) file. The name of each file is the name of its class.

Example: Class **Date** is defined in two files: **Date.h** and **Date.cpp**

All of the code developed for this application should be enclosed in the **`ama`** namespace.

MILESTONE 1: THE DATE CLASS

To kick-start this project, clone/download milestone 1 from the course repository and code the `Date` class.

The `Date` class encapsulates a date that is readable by an `std::istream` and printable by an `std::ostream` using the following format for both reading and writing: `YYYY/MM/DD`, where `YYYY` refers to a four-digit value for the year, `MM` refers to a two-digit value for the month and `DD` refers to a two-digit value for the day in the month.

Implement the `Date` class using following specifications below.

CONSTANTS:

In the `Date.h` header file, predefine the following constants as integers:

- `min_year` with the value 2019. This represents the minimum year acceptable for a valid date.
- `max_year` with the value 2028. This represents the maximum year for a valid date.
- `no_error` with the value 0. A date object with this status is either in a safe empty state or holds a valid date.
- `error_year` with the value 1. The client has attempted to set an invalid year for the date object. The object should be in a safe empty state.
- `error_mon` with the value 2. The client has attempted to set an invalid month for the date object. The object should be in a safe empty state.
- `error_day` with the value 3. The client has attempted to set an invalid day for the date object. The object should be in a safe empty state.
- `error_invalid_operation` with the value 4. The client has attempted to perform an invalid operation on the date object. The object should store the same date as before the invalid operation was attempted.
- `error_input` with the value 5. The date object failed to read data from an input steam (stream data was in the wrong format). The object should be in a safe empty state.

CLASS MEMBERS:

Add to the class attributes to store the following information:

- **Year**: an integer with a value between the limits defined above.
- **Month**: an integer with a value between 1 and 12.
- **Day of the Month**: an integer with a value between and the maximum number of days in the month. Use the function `int mdays(int year, int month)` to find out how many days are in a given month for a given year. Note that February can have 28 or 29 days depending on the year.
- **Status**: an integer used by the client to determine if the object contains a valid date. This attribute should have the value `no_error` or one of the `error_*` constants defined above.

Add to the class the following private functions:

- `void status(int newStatus)`: This function sets the status of the date object to the value of the parameter.
- `int mdays(int year, int month) const`: This function returns the number of days in a given month for a given year. Use the implementation below:

```
int Date::mdays(int year, int mon) const
{
  int days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31, -1 };
  int month = mon >= 1 && mon <= 12 ? mon : 13;
  month--;
  return days[month] + int((month == 1)*((year % 4 == 0) && (year % 100 != 0)) || (year % 400 == 0));
}
```

Add to the class the following public functions:

- A default constructor: sets the object to an empty state and the status to `no_error`. Use the date `0000/00/00` as the empty state.
- A custom constructor with three arguments: year, month and day (in this order). If the date specified by the arguments is correct (each number is within range), this constructor should store the arguments in the attributes and set the status to `no_error`.

If any of the parameters is not correct, the constructor should set the date object in a safe empty state and set the status to a value that

indicates which parameter was invalid (see the constants defined above).

The parameters should be checked in the following order: year, month, and then day.

- `status(...)`: A query that returns an integer representing the status of the current object.
- `clearError(...)`: A modifier that resets the status of an object to `no_error`. This function should not update the date stored in the object.

This function should return nothing.

- `isGood(...)`: A query that returns true if the object stores a valid date and is not in error mode.
- `operator+=(int days)`: A modifier that adds to the date stored by the object the number of days specified in the parameter.

If the current object is in error mode or in a safe empty state, this function does not update the stored date and changes the status of the object to `error_invalid_operation`.

If adding the number of days specified by the parameter would place the days attribute outside the acceptable limits, this function does not update the stored date and changes the status of the object to `error_invalid_operation`.

- `operator++()`: A modifier that adds one day to the date stored by the object (prefix). This function should return the updated current instance.

If the current object is in error mode or in a safe empty state, this function does not update the stored date and changes the status of the object to `error_invalid_operation`.

If adding one day would place the days attribute outside the acceptable limits, this function does not update the stored date and changes the status of the object to `error_invalid_operation`.

- `operator++(int)`: A modifier that adds one day to the date stored by the object (postfix). This function should return a copy of the instance before it gets updated.

If the current object is in error mode or in a safe empty state, this function does not update the stored date and changes the status of the current object to `error_invalid_operation`.

If adding one day would place the days attribute outside the acceptable limits, this function does not update the stored date and changes the status of the current object to `error_invalid_operation`.

- `operator+(int days)`: A query that adds to the date stored by the object the number of days specified in the parameter. The result is stored in a new object.

  If the current object is in error mode or in a safe empty state, this function returns a copy of the current object with the status set to `error_invalid_operation`.

  If adding the number of days specified by the parameter would place the days attribute outside the acceptable limits, this function returns a copy of the current object with the status set to `error_invalid_operation`.
- `operator==(const Date& rhs)`: A query that returns `true` if two date objects store the same date (does not check status of either object).
- `operator!=(const Date& rhs)`: A query that returns `true` if two date objects store different dates (does not check status of either object).
- `operator<(const Date& rhs)`: A query that returns `true` if the current object stores a date that is before the date stored in `rhs` (does not check status of either object).
- `operator>(const Date& rhs)`: A query that returns `true` if the current object stores a date that is after the date stored in `rhs` (does not check status of either object).
- `operator<=(const Date& rhs)`: A query that returns `true` if the current object stores a date that is before or equal to the date stored in `rhs` (does not check status of either object).
- `operator>=(const Date& rhs)`: A query that returns `true` if the current object stores a date that is after or equal to the date stored in `rhs` (does not check status of either object).
- `istream& Date::read(istream& is)`: A query that reads from an input stream a date in the following format: `YYYY?MM?DD` (three integers separated by a single character).

  This function does not prompt the user.

  If the reading fails at any point (when the reading fails, the function `is.fail()` returns `true`), this function sets the object in a safe state and updates the state attribute to `error_input`.

If this function reads the numbers successfully, and the read values are valid, it stores them into the instance variables and sets the status to `no_error`.

Regardless of the result of the input process, your function returns a reference to the `std::istream` object.

- `ostream& Date::write(ostream& os)`: A query that writes the date to an `std::ostream` object in the following format: `YYYY/MM/DD`, and then returns a reference to the `std::ostream` object.

Add to the following free helper function functions:

- `operator<<`: Prints the date to the first parameter (use `Date::write(…)`).
- `operator>>`: Reads the date from the first parameter (use `Date::read(…)`).

NOTE: You can add as many **private** members as your design requires. Do not add extra public members.

SUBMISSION

If not on matrix already, upload `Date.h`, `Date.cpp` and `ms1.cpp` to your matrix account. Compile and run your code and make sure everything works properly.

Then, run the following command from your account (use your professor's Seneca `userid` to replace `profname.proflastname`):

~profname.proflastname/submit 244_ms1<ENTER>

and follow the instructions.

IMPORTANT: Please note that a successful submission does not guarantee full credit for this workshop. If your professor is not satisfied with your implementation, your professor may ask you to resubmit. Resubmissions will attract a penalty.

MILESTONE 2: THE ERRORSTATE CLASS

The `ErrorState` class manages the error state of client code and encapsulates the last error message.

Any client can define and store an `ErrorState` object. If a client encounters an error, it can set its `ErrorState` object to an appropriate message. The client sets the length of the message—the message must be a **dynamically** allocated string managed by this class.

The `ErrorState` object reports whether or not an error has occurred. The **conversion to bool operator** reports if the object holds a message (an error has occurred). If an error has occurred, the object can display the message associated with that error using the **insertion operator** (`<<`).

> **NOTE:** This milestone does not use the `Date` class.

Implement the `ErrorState` class using following specifications below.

CLASS MEMBERS:

- Add to the class an attribute to store the address of the message, if any, in the current instance.

Add to the class the following public functions:

- `explicit ErrorState(const char* errorMessage = nullptr)`: No/One argument constructor. This function receives the address of a C-style null terminated string that holds an error message.

  If the address is `nullptr`, this function puts the object in the safe empty state (the object is in the empty state when it stores `nullptr` in the attribute representing the message).

  If the address points to an empty string, this function puts the object in the safe empty state.

  If the address points to a non-empty message, this function allocates memory for that message and copies the message into the allocated memory. **This function should not allocate more memory than it is necessary to store the string**.
- `ErrorState(const ErrorState& other) = delete`: This class should not allow copying of any `ErrorState` object.

- `ErrorState& operator=(const ErrorState& other) = delete`: This class does not allow copy assignment.
- `~ErrorState()`: This function de-allocates any memory that has been dynamically allocated by the current object.
- `operator bool() const`: this function return `true` if the current instance is storing a valid message (is not in the empty state).
- `ErrorState& operator=(const char* pText)`: stores a copy the text received in the parameter. This function handles the parameter in the same way as the constructor.

  This is an overload of the assignment operator—this function is **not the copy assignment** operator.

  Make sure that your code doesn't have memory leaks.
- `void message(const char* pText)`: stores a copy the text received in the parameter. This function handles the parameter in the same way as the constructor.

  Make sure that your code doesn't have memory leaks.
- `const char* message() const`: this query returns the address of the message stored in the current object.

  If the object is in the safe empty state, this function returns `nullptr`.

Add the following helper function:

- `operator<<`: Prints the text stored in an `ErrorState` object to a stream.

  If the object is in the safe empty state, this function does nothing.

NOTE: You can add as many **private** members as your design requires. Do not add extra public members.

NOTE: See that some of the functions above have similar functionality. Reuse the code instead of duplicating it!

SUBMISSION

If not on matrix already, upload `ErrorState.h`, `ErrorState.cpp` and `ms2.cpp` to your matrix account. Compile and run your code and make sure everything works properly.

Then, run the following command from your account (use your professor's Seneca `userid` to replace `profname.proflastname`):

~profname.proflastname/submit 244_ms2<ENTER>

and follow the instructions.

IMPORTANT: Please note that a successful submission does not guarantee full credit for this workshop. If your professor is not satisfied with your implementation, your professor may ask you to resubmit. Resubmissions will attract a penalty.

MILESTONE 3: THE PRODUCT CLASS

The `Product` class is a concrete class that encapsulates the general information for an AMA product.

Define and implement your `Product` class in the `ama` namespace. Store your class definition in a file named `Product.h` and your implementation in a file named `Product.cpp`.

> **NOTE:** This milestone does not use the `Date` class, but uses the `ErrorState` class.

In the `Product.h` header file, predefine the following constants as integers:

- `max_length_label` with the value 30. This is used when accepting data from user or displaying data to user.
- `max_length_sku` with the value 7. This represents the maximum number of characters in an SKU (stock keeping unit).
- `max_length_name` with the value 75. This represents the maximum number of characters for the name of the product.
- `max_length_unit` with the value 10. This represents the maximum number of characters in the user descriptor for a product unit.
- `write_condensed` with the value 0. This is used to signal what we want the data inserted into a stream in a condensed form.
- `write_table` with the value 1. This is used to signal what we want the data inserted into a stream in a table form.
- `write_human` with the value 2. This is used to signal what we want the data inserted into a stream in a human readable form.

In the `Product.h` header file, predefine the following constant as a floating point number in double precision:

- `tax_rate`: the current tax rate—13%.

PRODUCT CLASS MEMBERS:

Add to the class attributes to store the following information:

- A **constant** character that indicates the type of the product – for use in the file record. This attribute must be initialized with an initialization list; cannot be initialized in the body of a constructor.

- A character array that holds the product's SKU (stock keeping unit) – the maximum number of characters excluding the null byte is defined by a constant.
- A character array that describes the product's unit – the maximum number of characters excluding the null byte is defined by a constant.
- A pointer that holds the address of a C-style string in dynamic memory containing the name of the product. **This the resource that the class must manage!**
- An integer that holds the quantity of the product currently on hand (available).
- An integer that holds the quantity of the product needed.
- A double that holds the price of a single unit of the product before any taxes.
- A bool that identifies the taxable status of the product; its value is true if the product is taxable.
- An `ErrorState` object that holds the error state of the `Product` object.

Add to the class the following **protected** functions:

- `void message(const char* pText)`: This function receives the address of a C-style null-terminated string holding an error message and stores that message in the `ErrorState` object.
- `bool isClear() const`: This query returns `true` if the `ErrorState` attribute contains no error message; `false` otherwise.

Add to the class the following **public** functions:

- **Zero/One Argument Constructor**: This constructor optionally receives a character that identifies the product type. The default value is `'N'`. This function stores the character received in an attribute and sets the current object to a safe recognizable empty state.
- **A Custom Constructor with 7 arguments**: This constructor receives in its seven parameters values in the following order:
  - the address of an unmodifiable C-style null terminated string holding the SKU of the product

    - the address of an unmodifiable C-style null terminated string holding the name of the product
    - the address of an unmodifiable C-style null terminated string holding the unit for the product
    - a double holding the product's price before taxes – defaults to zero
    - an integer holding the quantity needed of the product – defaults to zero
    - an integer holding the quantity of the product on hand – defaults to zero
    - a Boolean value indicating the product's taxable status – defaults to true

  This constructor allocates enough memory to hold the name of the product. If the name parameter is null, this constructor sets the object in an empty state. If the name parameter is valid, this constructor stores all parameters in attributes. The type for the product is set to `'N'`.

- **The Copy Constructor**.
- **The Destructor**.
- **The Copy Assignment Operator**.
- `int operator+=(int cnt)`: This modifier receives an integer identifying the number of units to be added to the available quantity attribute and returns the updated number of units on hand. If the integer received is positive-valued, this function adds it to the quantity on hand. If the integer is negative-valued or zero, this function does nothing and returns the quantity on hand (without modification).
- `bool operator==(const char* sku) const`: This query returns `true` if the string specified in the parameter is the same as the string stored in the SKU attribute of the current instance; `false` otherwise.
- `bool operator> (const char* sku) const`: This query returns `true` if the SKU attribute from the current instance is greater than the string stored at the received address (according to how the string comparison functions define 'greater than'); `false` otherwise.
- `bool operator> (const Product&) const`: This query returns `true` if the name of the current object is greater than the name of the

`Product` received as parameter object (according to how the string comparison functions define 'greater than'); `false` otherwise.

- `int qtyAvailable() const`: This query returns the value of the attribute storing how many units of product are available.
- `int qtyNeeded() const`: This query returns the value of the attribute storing how many units of product are needed.
- `const char* name() const`: This query returns the address of the name of the product.
- `double total_cost() const`: This query returns the total cost of all available units of product, including tax.
- `bool isEmpty() const`: this query returns `true` if the object is in the empty state; `false` otherwise.
- `std::istream& read(std::istream& in, bool interractive)`: This function reads data from the stream, and stores it in the attributes. Depending on the second parameter, this function prompts the user asking for values, or doesn't interact with the user.

If the second parameter is `false`, this function extracts all the information from the stream and doesn't interact with the user. The format of the expected data is the following:

```
SKU,NAME,UNIT,PRICE,TAX,QTY_AVAILABLE,QTY_NEEDED<ENDL>
```

That is a sequence of fields, separated by comma. After the last field, there might be a `'\n'` or the end of the stream.

If the second parameter is `true`, this function interacts with the user, asking for data in the following order:

```
                 Sku: <User Types Here - a C-style string>
Name (no spaces): <User Types Here - a C-style string>
                Unit: <User Types Here - a C-style string>
    Taxed? (y/n): <User Types Here - y, Y, n, or N are acceptable>
               Price: <User Types Here - a number>
Quantity on hand: <User Types Here - an integer>
 Quantity needed: <User Types Here - an integer>
```

The text prompted to the user will be displayed in a field of size `max_length_label`, aligned to the right. The field size includes the space (`␣`) after the colon (`:`).

If there are errors while reading a field, this function stops asking data from the user, sets the failure bit of the first parameter (by calling `istr.setstate(ios::failbit)`), and sets the `ErrorState` attribute to one of the following messages, depending on which field encountered the issue:

```
Only (Y)es or (N)o are acceptable!
Invalid Price Entry!
Invalid Quantity Available Entry!
Invalid Quantity Needed Entry!
```

This function accepts the data **only if there are no errors reading it**.

- `std::ostream& write(std::ostream& out, int writeMode) const`: This function writes the content of the current instance in the stream received as the first parameter.

If the object contains an error message, this function prints the error message and exits

If the current instance is in empty state, this function is not doing anything.

If the `writeMode` is set to `write_condensed`, this function writes the content of the current instance in the following format:

```
TAG,SKU,NAME,UNIT,PRICE,TAX,QTY_AVAILABLE,QTY_NEEDED<ENDL>
```

If the `writeMode` is set to `write_table`, this function writes the content of the current instance in the following format (␣ marks a mandatory blank space, not part of the field width):

```
␣SKU␣|␣NAME␣|␣UNIT␣|␣PRICE␣|␣TAX␣|␣QTY_AVAIL␣|␣QTY_NEEDED␣|
```

Field	Size	Alignment	Observations
SKU	`max_length_sku`	Right	
Name	16	Left	If the name is more than 16 characters, write only 13 characters, followed by three dots (…)
Unit	10	Left	
Price	7	Right	Precision is set to 2
Tax	3	Right	Writes “yes” or “no”
Quantity Available	6	Right	
Quantity Needed	6	Right	

If the `writeMode` is set to `write_human`, this function writes the content of the current instance in the following format:

```
                  Sku: ␣SKU<ENDL>
                 Name: ␣NAME<ENDL>
                Price: ␣PRICE<ENDL>
    Price after Tax: ␣PRICE_WITH_TAX<ENDL>
Quantity Available: ␣QTY_AVAILABLE ␣UNIT<ENDL>
   Quantity Needed: ␣QTY_NEEDED ␣UNIT<ENDL>
```

The labels are displayed in a field of size `max_length_label`, aligned to the right. The field size includes the space (`␣`) after the colon (`:`).

NOTE: You can add as many **private** members as your design requires. Do not add extra public/protected members.

NOTE: Some of the functions above change the name of the product; make sure that there are **no memory leaks** when that happens.

NOTE: Reuse the code whenever possible. **Do not duplicate functionality!**

NOTE: With the exception of the tag field, what the function `write(...)` writes in condensed mode, the function `read(...)` should be able to extract in non-interactive mode.

UTILITIES

In another module, called `Utilities`, add the following *helper functions*, in the `ama` namespace (prototypes in the header, implementations in a source file):

- `double& operator+=(double& total, const Product& prod)`: Adds the total cost of the product into the first parameter and returns the result;
- `ostream& operator<<(ostream& out, const Product& prod)`: Writes into the first parameter, in a human readable format, the content of the second parameter.
- `istream& operator>>(istream& in, Product& prod)`: Reads from the first parameter a product in interactive mode.

SUBMISSION

If not on matrix already, upload `Utilities.h`, `Utilities.cpp`, `Product.h`, `Product.cpp`, `ErrorState.h`, `ErrorState.cpp` and `ms3.cpp` to your matrix account. Compile and run your code and make sure everything works properly.

Then, run the following command from your account (use your professor's Seneca `userid` to replace `profname.proflastname`):

~profname.proflastname/submit 244_ms3<ENTER>

and follow the instructions.

IMPORTANT: Please note that a successful submission does not guarantee full credit for this workshop. If your professor is not satisfied with your implementation, your professor may ask you to resubmit. Resubmissions will attract a penalty.

MILESTONE 4: THE iPRODUCT INTERFACE

The `iProduct` class is an interface that exposes the *Product* hierarchy to client applications. This class contains only pure virtual functions and cannot be instantiated.

Save your definition of the `iProduct` interface in a header file named `iProduct.h`. The interface should be defined in the `ama` namespace.

The definition of your `iProduct` interface includes the following **pure virtual** member functions (these functions will be implemented in the derived classes):

- `ostream& write(ostream& os, int writeMode) const`: This function writes the content of the current instance in the stream received as the first parameter. The second parameter signals the format of the output (see class `Product` for details).
- `istream& read(istream& is, bool interractive)`: This function reads data from the stream, and stores it in the attributes. Depending on the second parameter, this function prompts the user asking for values, or doesn't interact with the user (see class `Product` for details).
- `bool operator==(const char* sku) const`: This query returns `true` if the string specified in the parameter is the same as the SKU of the current instance; `false` otherwise.
- `double total_cost() const`: This query returns the total cost of all available units of product, including tax.
- `int qtyNeeded() const`: This query returns the how many units of product are needed.
- `int qtyAvailable() const`: This query returns the how many units of product are available.
- `const char* name() const`: This query returns the address of the string storing the name of the product.
- `int operator+=(int qty)`: This modifier receives an integer identifying the number of units to be added to the available quantity of product and returns the updated number of units on hand.
- `bool operator>(const iProduct& other) const`: This query returns `true` if the name of the current products is greater than the name of

the `iProduct` received as parameter (according to how the string comparison functions define 'greater than'); `false` otherwise.

CHANGES IN *PRODUCT* MODULE

Update the *Product* module by doing the following:

- Include the `iProduct.h` header in `Product.h`.
- Change the `Product` class definition to publicly inherit from the interface `iProduct`.
- Update the prototype of the `operator>` to accept as parameter an un-modifiable reference to `iProduct`: `bool operator>(const iProduct& other) const` (make sure to update the prototype in the `Product.cpp` as well; do not change the implementation).
- Move all the constants you have defined in `Product.h` to `iProduct.h` (still in the `ama` namespace).

NOTE: See that the functions from the interface are already implemented in the `Product` class.

CHANGES IN *UTILITIES* MODULE

Modify the prototypes of the three helper operators from this module to accept as parameter references to `iProduct` instead of references to `Product`, like this:

- `double& operator+=(double& total, const iProduct& prod)`
- `ostream& operator<<(ostream& out, const iProduct& prod)`
- `istream& operator>>(istream& in, iProduct& prod)`

NOTE: The implementation of these operators should not change.

Declare and define the following function:

- `iProduct* createInstance(char tag)`: This function is responsible to dynamically create instances in the *Product* hierarchy.

If the parameter has the value `'N'` or `'n'`, this function should **dynamically** create an object of type `Product` using the default constructor and return its address.

If the parameter has any other value, this function should return null.

SUBMISSION

If not on matrix already, upload `Utilities.h`, `Utilities.cpp`, `iProduct.h`, `Product.h`, `Product.cpp`, `ErrorState.h`, `ErrorState.cpp` and `ms4.cpp` to your matrix account. Compile and run your code and make sure everything works properly.

Then, run the following command from your account (use your professor's Seneca `userid` to replace `profname.proflastname`):

~profname.proflastname/submit 244_ms4<ENTER>

and follow the instructions.

IMPORTANT: Please note that a successful submission does not guarantee full credit for this workshop. If your professor is not satisfied with your implementation, your professor may ask you to resubmit. Resubmissions will attract a penalty.

MILESTONE 5: THE `PERISHABLE` CLASS AND *AMA* APPLICATION