MET cS665 Summer 2 2018

bob

Abel Gancsos productions | Hawthorne, NJ

AMGVending

2018

Contents

Introduction 2

1 Application Description 3

1.1 Implementation Details 3

1.2 Assumptions 3

1.3 Prerequisites 3

2 Assignment Tasks 4

2.1 Implementation Description 4

2.2 UML Class Diagram 5

2.3 UML Sequence Diagram 6

Bibliography 7

Appendices 8

Appendix A – Setup 8

Appendix A – Building project 8

Appendix B – Running executable 9

# Introduction

As more and more retail experiences becoming more automated, HotDrinks Brewing Company of New Jersey is looking towards joining this movement. They have been offering their brewed beverages to the masses by having an onsite barista at many locations; however, they now see the inefficiencies of this approach. They are looking to design an automated vending machine that can provide their current drink offers to patrons, use their proprietary brewing mechanism, and then provide the same drink to customers without having a barista at their location. They already have the hardware designed; however, they now need the software controller that makes the magic happen.

Some requirements of the controller is that it should offer the same drinks as they currently have, such as Coffee Espresso, Coffee Americano, Coffee Latte Macchiato, Black Tea, Green Tea, and Yellow Tea. If this concept works for them, they may add other options as they see fit. The drinks should also be customizable with condiments, such as sugar or milk. For now, condiments will be limited to a quantity of 3 each, so that we can avoid issues brought on by user input. This limit might eventually be raised or lifted completed; however, this is a requirement for now.

# 1 Application Description

## Implementation Details

*The following details are also available in the README.md file*

The following utility, AMGVending, acts as a controller for a new fully automated beverage vending machine. This is to say that run in interactive mode, the utility should be prepared for a customer by having an available prompt, in this case in a shell command window. Once a patron arrives and is ready to order, they select from the menu options the type of drink they are interested in. The controller then prompts for the subtype or sort of drink the patron would like to have, and then it prompts the patron if they would like any toppings or condiments. If the patron selects to add condiments, the controller then displays a menu option for available condiments. The vending machine then prompts the patron for quantity of the condiment, which must be within the valid scope. Then vending machine then prepares the drink and dispenses it to the patron. The controller then goes back to the main menu for the next transaction.

In order to handle future drink types, the controller uses abstract classes for the drink objects. As for the drink sorts, each derived class of the abstract drink type comes with a sort enum, which will be primarily used for input validation. As for the condiments, we use another abstract class as we suspect that additional condiments might be added in the future.

The controller then uses a "session" class as a wrapper to build out the transaction, which holds details of the selected options.

## 1.2 Assumptions

* The dispensing mechanism works or the software group is not responsible for any mechanical bugs.
* Maybe some day the vending machine is to be run in a batchable mode, therefore, certain switches or flags are built into the controller.
* Additional drink types will be added in the future.
* Additional drink sorts will be added in the future.
* Additional drink condiments will be added in the future.
* The restriction on drink condiment quantity may be raised or lifted completely.
* The payment has already been provided before the controller is used.

## 1.3 Prerequisites

* C++ compiler with C++11 capability
* CMake version 3.11 or higher
* XCode command-line tools (MacOS only)
* Visual Studio 2017 or higher (Windows only)

# 2 Assignment Tasks

## 2.1 Implementation Description

*The following details are also available in the README.md file*

This utility was implemented using abstract classes due to their many benefits towards the functionality of the project. For example, because it is expected that there will be new drink types added, an abstract class, VendingDrink, was created to manage the base structure. This base structure would then be inherited by any new drink type, for which certain operations would be implemented using polymorphism in order to provide appropriate results for the new drink. So in order to add a new drink type or remove an existing one, the class simply needs to implement the abstract methods or the class should simply be removed.

An abstract class not only serves as a class which child classes will inherit the base characteristics, but also provides a way to dynamically iterate through a collection of drink objects by calling the abstract methods. When the abstract method is called against the base class within the iteration, what actually happens is that the implementation is called against the child class. This is possible since VendingDrink cannot be instantiated, only through the child class that implements the abstract methods. This then provides the actual drink type, but represented as the abstract VendingDrink object.

Another design approach that was used was helper classes. Although this is a very basic concept, by using the helper class, AMGVendingHelpers, this offered a way to call common blocks of code in different areas of the program. By having a central location where these methods could be called from, duplication of code can be avoided.

One aspect of my strategy can be seen in the session/wrapper/client class, AMGVending, especially where the user is prompted for menu options. There is a collection of drink type names and drink condiment names, which are used as a mapping for when prompting the user for those options by iterating through them and asking the user for the numeric menu value. This numeric menu value is then mapped to the collection index. By using this strategy, there is no reason for conditional statements nor switch statements, the value is simply taken by the index number. The name retrieved by the collection index is then used to instantiate the concrete instance of the abstract classes. The reason why this strategy is beneficial for the utility is because it provides the capability of dynamically listing menu options and then building out concrete instances without the need of large if-else blocks.

When it comes to simplicity and understandability, the utility attempts to address both concerns by using small methods. When the methods started to get quite large, it was hard to follow, so the large method was broken down into smaller methods which could be better understand and more easily read. Although there was no hard limit of the number of lines in the methods nor the number a methods a class could have, methods were subjectively broken down into smaller pieces based on the operations that were being performed. For example, the AMGVending.PrintMenu() method was originally made up of all the prompt methods, which cased the PrintMenu() method to become bloated. After breaking up the operations, it was easier to understand what the PrintMenu() method was trying to get accomplished. In addition, every single method was commented and any strange looking operation was noted to explain the purpose.

One thing to note, not really a design thing, is that JavaDoc style comments were used. This was because C++ doesn’t really offer a nice way to document the source code, such as JavaDocs or XML comments in C#. JavaDocs was used as it offers a simpler and more understandable format, which many developers may already be familiar with.

## 2.2 UML Class Diagram



## 2.3 UML Sequence Diagram



# Bibliography

Döring, A. (2016, 11 07). *minimal\_cmake\_example*. Retrieved 07 03, 2018, from GitHub Web site: https://github.com/krux02/minimal\_cmake\_example

Google Corporation. (2018, 06 19). *Google C++ Style Guide*. Retrieved 07 03, 2018, from Google GitHub styles: https://google.github.io/styleguide/cppguide.html

Kitware Corporation. (n.d.). *Download*. Retrieved 07 03, 2018, from CMake Web site: https://cmake.org/download/

Lucid Software Corporation. (2017). *Class Diagram Tutorial*. Retrieved 7 25, 2017, from Lucid Software Corporation Web site: https://www.lucidchart.com/pages/uml/class-diagram

Microsoft Corporation. (2010). Visio. (2010). Redmond, WA, USA.

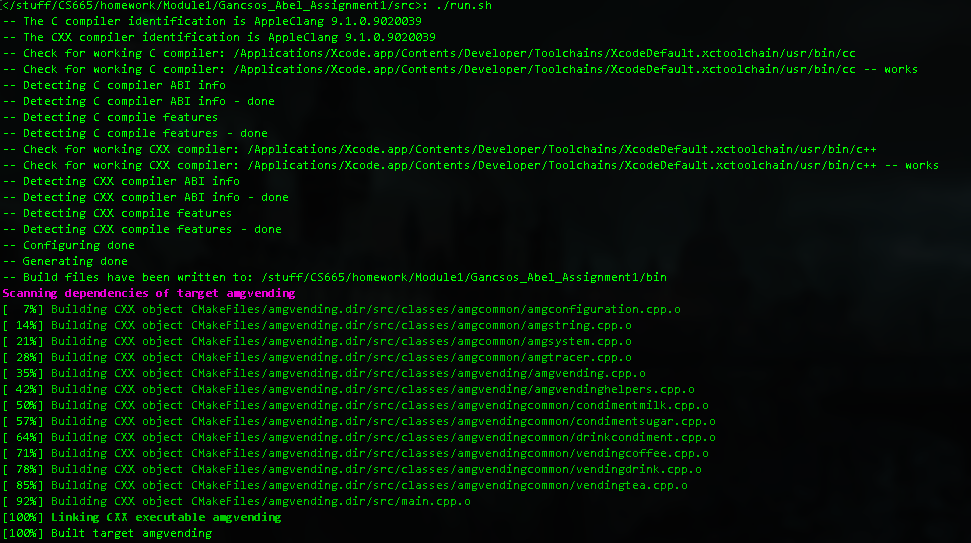
Oracle Corporation. (2017). VirtualBox. (5.1.18). Redwood City, CA, USA.

# Appendices

## Appendix A – Setup

The project can be build using the run.sh or run.bat scripts, depending on the platform. Both of these scripts will run cmake to generate the CMakeFiles and then run the make command in the bin directory. Alternatively, the project can be built manually using the CMakeLists.txt file that comes with the package.

## Appendix A – Building project



## Appendix B – Running executable

