MET cS665 Summer 2 2018

Abel Gancsos

Abel Gancsos productions | Hawthorne, NJ

AMGDeliveryDispatch

2018

Contents

[1 Application Description 3](#_Toc519352359)

[1.1 Implementation Details 3](#_Toc519352360)

[1.2 Product Types 3](#_Toc519352361)

[1.3 Requirements 3](#_Toc519352362)

[1.4 Assumptions 3](#_Toc519352363)

[1.5 Prerequisites 4](#_Toc519352364)

[2 Assignment Tasks 5](#_Toc519352365)

[2.1 Implementation Description 5](#_Toc519352366)

[2.2 UML Class Diagram 6](#_Toc519352367)

[2.3 UML Sequence Diagram 7](#_Toc519352368)

[Bibliography 8](#_Toc519352369)

[Appendices 9](#_Toc519352370)

[Appendix A – Setup 9](#_Toc519352371)

[Appendix A – Building project 9](#_Toc519352372)

[Appendix B – Running executable 10](#_Toc519352373)

# 1 Application Description

## Implementation Details

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

This utility, AMGDeliveryDispatch, is a system that helps assign delivery orders to appropriate vans based on certain criteria once the purchase has been completed. The system starts by taking in a certain number of orders containing various number of items from different customers at different shops. The assignment process then scans the type of items in the orders, checks if it's the customers birthday, calculates the total distance of the delivery, and then checks if there's an available driver. To ensure appropriate delivery time and proper state of the product upon delivery, the assignment process also checks for any traffic that could occur while in transit. If an order is not able to be dispatched due to lack of drivers or unavailable resources, the order waits until a proper resource has become available. Once the driver has been assigned, the customer is then notified of a scheduled delivery. The controller then continues to recalculate the assignment at an interval until all orders have been delivered.

## Product Types

\* Food

\* Cold

\* Warm

\* Flower Bouquets

\* Gift chocolate boxes

## 1.3 Requirements

* If a cold meal is to be delivered more than 2 miles away, it must go to a van with a freezer.
* If it's a customers birthday and they have an order, they must receive a gift chocolate box.
* When there is a lot of traffic, ALL cold foods must go to vans with freezers.

## Assumptions

* The order has already been completed and paid for by the time the process enters into this utility.
* In addition to notifying the customer of an expected delivery, the driver is told where to pick the order up.
* Although the system would use GPA coordinates to calculate driver location, the distance is calculated by random numbers in miles for simulation purposes.
* More product types can be added in the future.
* Food types can be of any type.
* Flower bouquets can include any flower combination.
* Chocolate boxes may contain different chocolate types.
* Drivers and products are predefined and the user doesn't control this data.
* One tick in the simulation is the same as travelling one mile in distance.
* An order can have multiple items, but no more than one item of the same type.
* The total distance of a delivery is considered the distance of the driver from the store plus the distance of the customer from the store.
* An order doesn't become a delivery until it is ready for dispatch.

## 1.5 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*

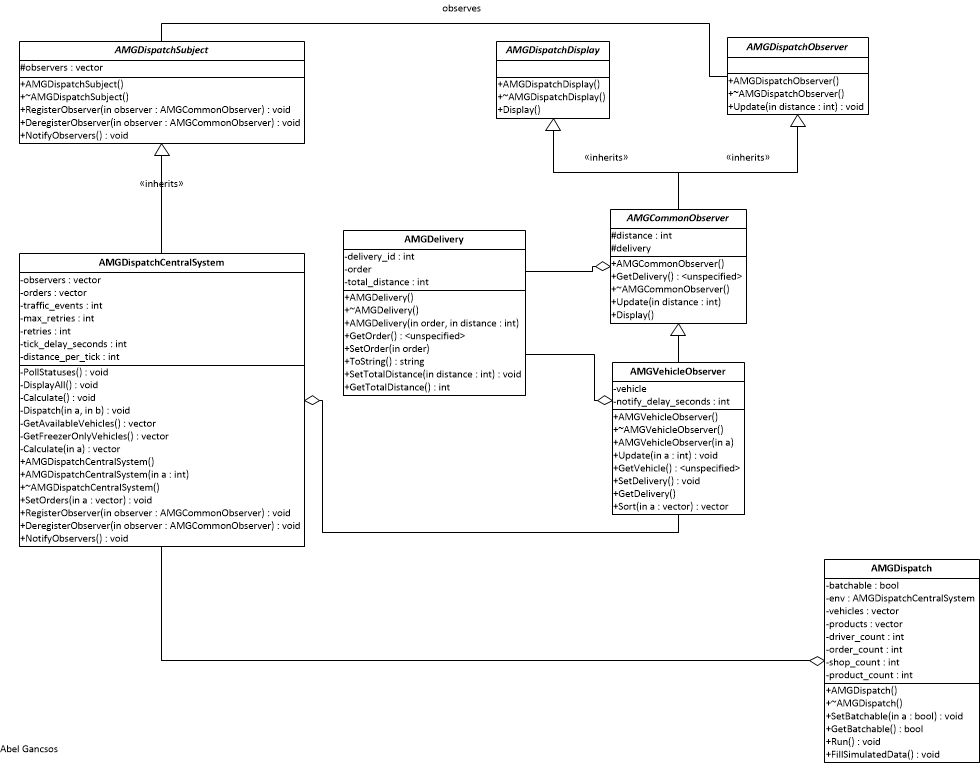
In brief, because this use case requires data to be spread across many different instances (vehicles), while maintaining a constant access to the code for consistency, the Observer design pattern was used. By having a single subject which many observer objects can monitor, the data can be manipulated and displayed in different ways without the main system or subject missing a change to the data. Because this system also requires several objects to be created, an ideal implementation would use a factory method pattern or even an abstract factory for their ease of use when creating objects at runtime. One of these patterns; however, was not used, but would be in a future release.

In regards to flexibility, because abstract classes and polymorphism is prominent in the implementation, it is quite easy to add new types such as products, subjects (observies), people, and observers. If there is ever a need to have a different implementation for specific product types, a new class can simply be created from the AMGProduct parent class with the new implementation while keeping the functionality of the main system. Another great example of how the implementation used is flexible is the fact that many different types of observers can be created from the base AMGDispatchObserver class. Granted the naming indicates that it’s specific to this system, the design takes into consideration the possible need for other observer types because it might not always be a vehicle that’s observing the data in the system.

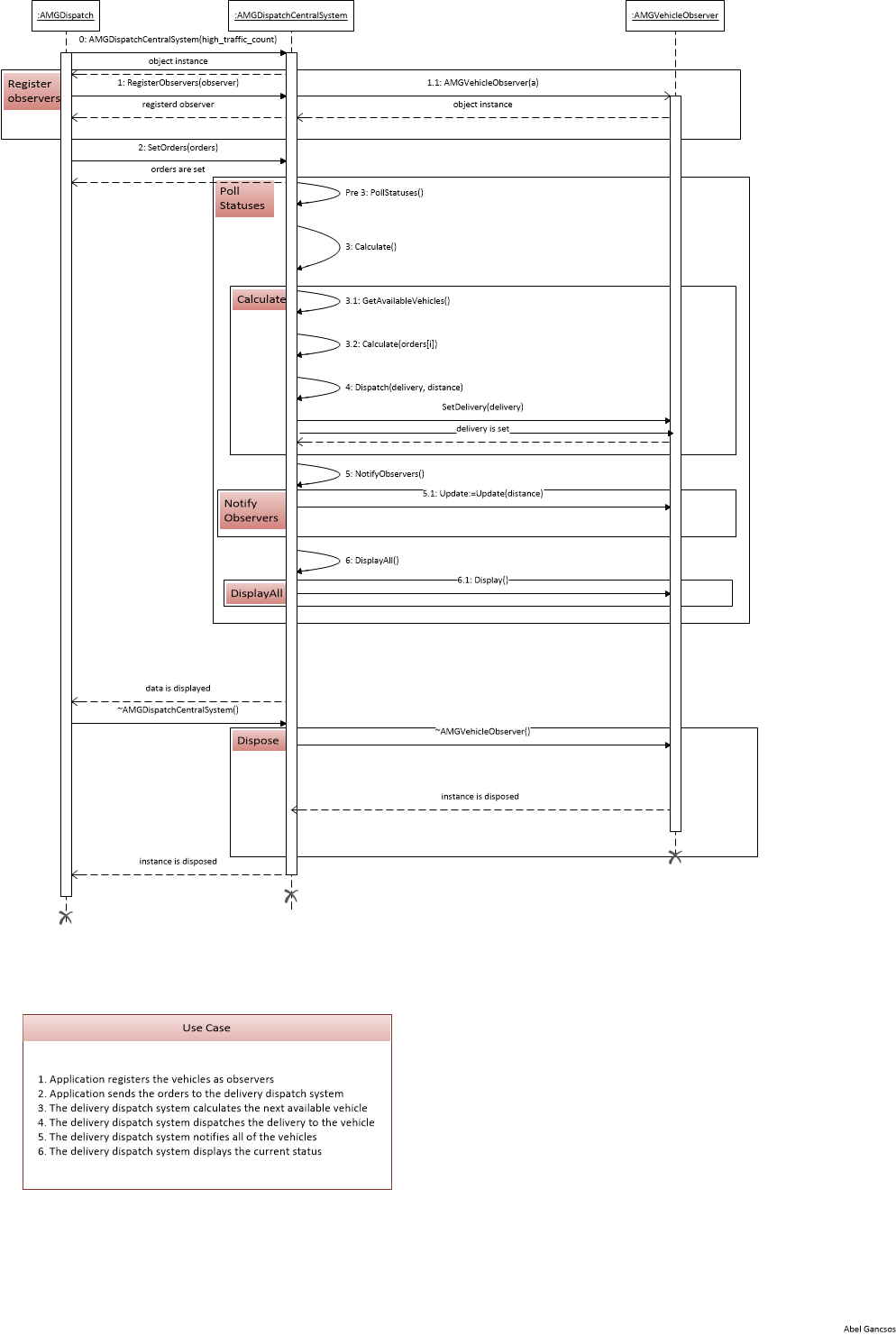
As for simplicity, although the implementation looks large at first, further explanation might help to show otherwise. Let’s start by reviewing the organization or packaging of the classes. There’s the AMGCommon package, which contains a scattered set of classes that are included in most of my projects. This package offers common and known methods that help spawn utilities quicker. Then there’s the AMGDispatchCommon, which holds the common object types in the dispatch system. For example, this is where the AMGVehicle, AMGDelivery, AMGOrder, and AMGItem classes are located, which are also important classes for the dispatching system overall. The benefit of this package is that if any change that was made in this area of the code breaks another part, then the change can easily be tracked and mitigated. On the other hand, because these common objects are held in one area, the chances of other portions of the code breaking might be higher; however, because these are common objects of which the design has been agreed upon, the changes will not be frequent. Then there’s the AMGDispatch package which simply holds classes that configure the environment and because this implementation is simulated, it also acts as a user in the form of “the Application”. So far, the classes in the mentioned packages exist either to make the implementation easier or safer. Finally, we have the two packages, AMGDispatchSubject and AMGDispatchObserver, where the bulk of the critical implementation is done. When a closer look is taken of these packages, one might see that they’re actually full of abstract classes used for adding different types of observers or subjects. In brief, the two classes without which this system would be without are the AMGDispatchCentralSystem and the AMGVehicleObserver classes. The AMGDispatchCentralSystem class is primarily used for calculating and managing the dispatching of deliveries, which the AMGVehicleObserver class is used to manage and display the data. Another note that can be made is that every method and every class is documented within the source code.

Finally, to avoid duplicate code, blocks that might be used multiple times were created as a method in an appropriate class that would require the least amount of dependencies. For example, because there is a constraint on the system that the vehicle with the lowest distance should be picked up first, a custom sort method would be implemented in the AMGVehicle class; however, because what’s actually being asked is the observer with the lowest distance vehicle, the method was implemented in the AMGVehicleObserver class. One note that should be made is that although several classes have the distance attribute, the value used here is dependent on the context, such as the object and what is really being asked for.

## 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.

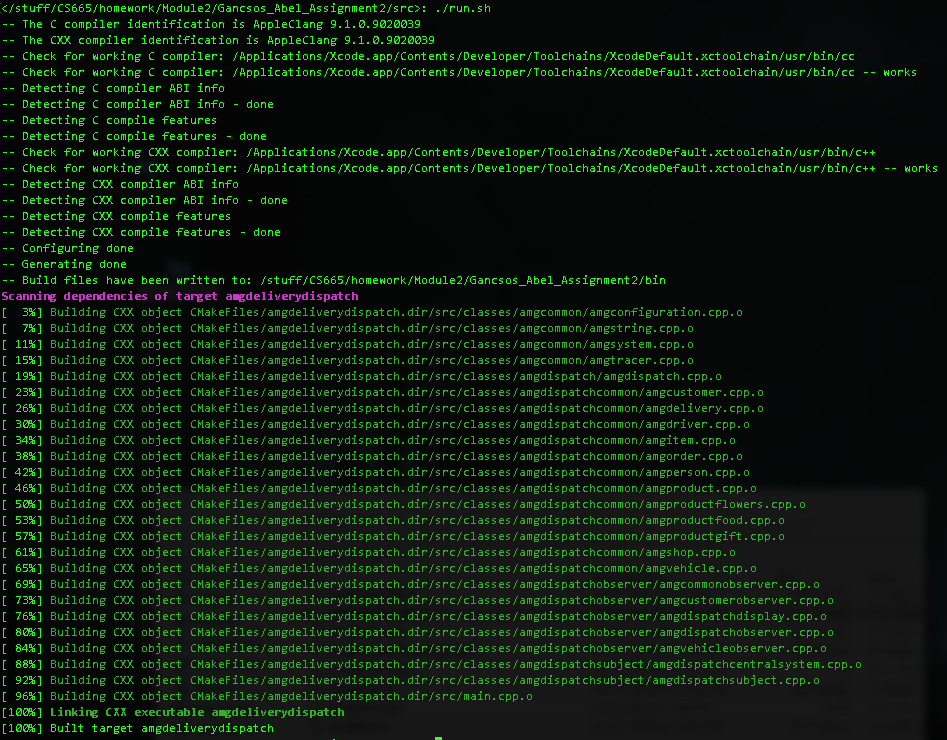
Oracle Corporation. (2017). VirtualBox. (5.1.18). Redwood City, CA, USA.

# 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

