# Task 1: Implementation Description

* The approach implemented is a slightly modified Observer Pattern.
  + The PublisherBase Interface is syndicate/SyndicateRequestor
  + The Concrete Publishers are the syndicate/Store
  + The SubscriberBase Interface is distribution/DriverBase
  + The concrete subscribers are distribution/Driver
  + The modification to the Observer Pattern in the additional distribution/Dispatcher which chooses from the list of subscribers (Drivers) and assigns one to the delivery request made by the Store.

## How is the flexibility, of your implementation, e.g., how you add or remove in future new

## types?

Being that the Observer Pattern is fairly closely adhered to, this design satisfies

the characteristic of loose coupling which implies great flexibility. In order to add

Publishers (classes like Store which would need to notify a Dispatcher) all we would need

to do is implement the PublisherBase interface and implement the methods therein. Similarly,

in order to add more Subscribers, let's say for example we wanted to loop the Customer into

the equation such that they could receive status of their order, all we need to do is

implement the DriverBase interface (which we might consider renaming to suggest a more

general usage as is implied by the Observer Pattern of "SubscriberBase"). In this way

a Customer could simply .updateSelf() accordingly.

## How is the simplicity and understandability of your implementation?

The Application as implemented here is using #1) a pattern that is quite common in modern

software applications such as all social media, Uber/Lyft, AmazonPrime, Weather Applications

and the list is seemingly endless truly. In this way, it seems quite easy to relate this

system of an entity that creates an "order" and "notifies" a "subscriber base" as needed with

the additional feature of having a "middle man" Dispatcher who chooses which subscriber

makes the delivery. We might imagine expanding this idea to incorperating the idea that

an active driver cannot recieve an assignment from the dispatcher, or even more interesting

incorperating the idea of a route and in general the idea of location such that each driver

has a location and the dispatcher assigns drivers based on that instead of at random. These

concepts and additional ideas are so common in the current marketplace that it seems their

understandability and perhaps more aptly, their relatability, is quite high. To this end,

there are quite few moving parts in this pattern and implementation such that the simplicity

and elegance seems to be high as well.

## How you avoided duplicated code?

Running the risk of redundancy, simply by adhering quite strictly to the Observer Pattern

and deviating only to satisfy the additional requirements of the assignment such that

only one Driver is assigned per order. By adhering to the pattern there is virtually no

repetition in code and yet the preconditions are elegantly satisfied thusly exposing the

power of the pattern in the context of the outlined requirements.

# Task 2 – UML Diagrams

## AtYourDoor Class Diagram

1. AtYourDoor Class Diagram also available [here](https://lucid.app/lucidchart/e3aab99e-fd06-48c9-8c4e-86e0511f15fe/edit?beaconFlowId=38C88C7109BA4BA3&invitationId=inv_39996e16-49c8-4343-b6e6-9731ea87f277&page=0_0%23)

Diagram

Description automatically generated

## AtYourDoor Sequence Diagram

2. AtYourDoor Sequence Diagram also available [here](https://lucid.app/lucidchart/b6479242-5a55-4f44-902c-6376d81bf4e5/edit?beaconFlowId=E27F7A31F7F2F632&invitationId=inv_06f1ad90-c7f5-4f62-aa19-0c88d4fae214&page=0_0%23)

Diagram

Description automatically generated

# Task 3: Code

<https://github.com/metcs/met-cs665-assignment-2-MichaelKramerGuitar>