The Home Task should be done using Typescript.

**Step 1: Starting Simply**

You are coding up the business logic for a Package Handling system. Another team is creating a graphical user interface (GUI), but this is not ready yet. Yet another team is working on a persistence layer which, among other things, will be able to generate unique Shipment ID's for you when needed (see below), but this is also not yet ready.

You will need, at minimum, two classes: Client and Shipment. One will represent the controlling object that will interact with the GUI when it is available. The other will represent the object being shipped.

State object that will come from the end user via the GUI will have following interface:

* shipmentId (a **number** that represents an existing ID, or 0, which means you must generate a new, unique ID at construction time)
* toAddress (a **string** containing street, city, and state) – should be changeable
* fromAddress (a **string** containing street, city, and state) – should be changeable
* toZipCode (a **string** containing exactly 5 characters) – should be changeable
* fromZipCode (a **string** containing exactly 5 characters) – should be changeable
* weight (a **number**, storing the weight of the item in ounces)
* marks (an optional **string array** – will represent additional characteristic of shipment) – should be changeable

The client object will obtain a single instance of the object which represents the item being shipped, and then ask it to "ship itself". The return from this request will be a single **string** indicating the shipment ID, where the item was sent from, where it is going, and how much the cost was.

The cost will be determined by the weight. The rate of 39 cents per ounce will be applied.

Once the client has this return, it should send it to the console for display.

*Things to note:*

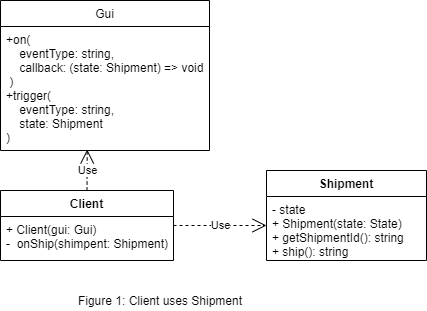
• Make sure you follow good practices. Always program by intention, and always encapsulate the construction of instances.

• Since you do not have the actual GUI to call upon for the state, you will have to dummy that up somewhere or you may create MockGui class that will be used for tests. Just make sure it is somewhere that is easy to change (hint: remember programming by intention tends to produce cohesive methods, which are easier to change and move from place to place).

• Since you don't have the actual persistence layer which can generate unique ID's for you, you'll need to simply generate a unique ID for now. Again, make sure this is as easy as possible to change when the real code is ready. Hint: make a method called **getShipmentID** which, for now, just increases a static int by one and returns it but in the future will get the real shipment ID.

• On the Figure 1 Gui is shown only for references. You may create a mock based on the Gui UML.

Figure 1 illustrates this in the UML.



**Step 2: Multiple Shippers**

Now you are informed that there are multiple partner companies used to actually ship the item in question. This will affect the per-ounce rate used when cost is determined.

The companies are:

* Air East: Based in Atlanta
* Chicago Sprint: Based in a suburb of Chicago
* Pacific Parcel: Based in San Diego

The correct shipper to use is determined by the zip code of the sender (the "from" zip code). If the from zip code begins with 1, 2, or 3, then the source location is in the east, and Air East will be used. A 4, 5, or 6 as the first digit of the from zip code means Chicago Sprint is the right choice. A 7, 8, or 9 will mean the western US, and therefore Pacific Parcel.

Air East charges 39 cents per ounce, as we are accustomed to paying (they are the vendor we have been using all along).

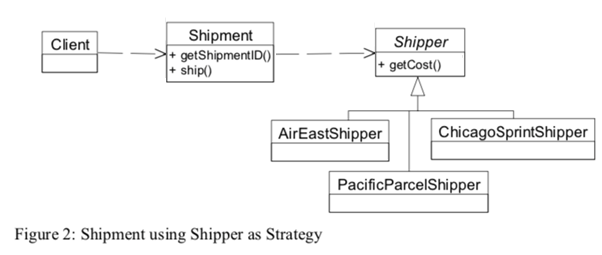
Chicago Sprint charges 42 cents.

Pacific Parcel charges 51 cents. Things are expensive in southern California!

Note that the "to" zip code does not affect the price. All our vendors have reciprocal agreements for cross-country shipping.

If the zip code is unknown, Air East will be the default.

Figure 2 illustrates this in the UML.



**Step 3: Different Kinds of Shipments**

Now there are three types of items being shipped: Letters, Packages, and Oversized. The way you know which type of thing is being shipped is determined by weight.

Regardless of which shipper is being used, the following rules apply:

A Letter is anything up to and including 15 ounces (less than a pound)

A Package is anything up to and including 160 ounces

An Oversized package is anything heavier than 160 ounces

The various shippers have different actions to take, depending on what sort of item is being shipped:

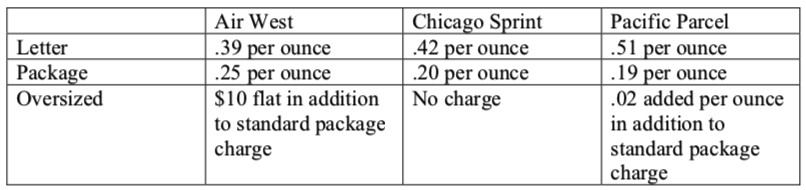
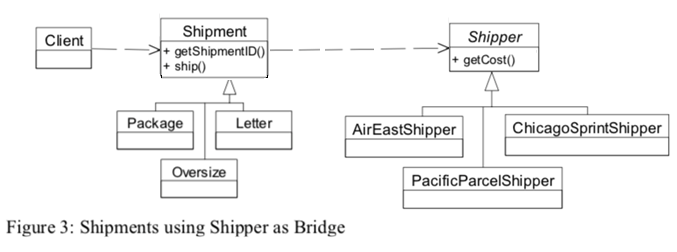


Figure 3 illustrates this in the UML.



**Step 4: Marking Shipments with Special Codes**

If selected in the GUI, the Shipment may be marked Fragile, Do Not Leave, and Return Receipt Requested, or any combination of these three. Here again, the GUI is not actually available, so you'll need to stub this out somehow, somewhere.

The effect of this will be that when the Client asks the Shipment to "ship itself", this extra information will appear at the bottom, in all uppercase and surrounded by double asterisks.

Here is an example, with all three marks applied:

*Shipment with the ID 17263 will be picked up from 12292 4th Ave SE, Bellevue, Wa 92021 and shipped to 1313 Mockingbird Lane, Tulsa, OK 67721*

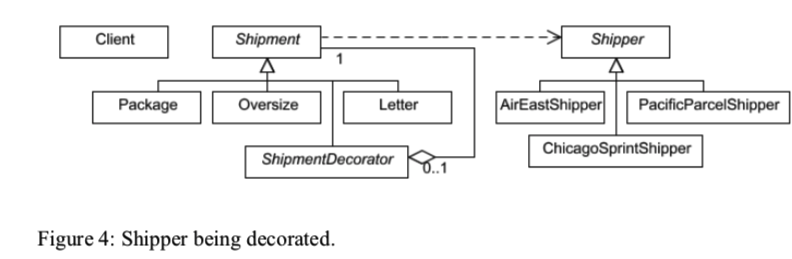
*Cost = 5.1*

*\*\*MARK FRAGILE\*\**

*\*\*MARK DO NOT LEAVE IF ADDRESS NOT AT HOME\*\**

*\*\*MARK RETURN RECEIPT REQUESTED\*\**

Figure 4 illustrates this in the UML.



**Important Notes:** you may not strictly follow every UML-diagram. If you want to replace some design pattern to another one, which you think fit better, or you suppose some pattern is redundant here you may not use it (please, add comments why it’s redundant). In case of implementation Shipment and Shipper classes as Singletons you may face with issue where Singleton can’t play role of Base/Abstract class. In such case you can consider using composition instead of inheritance OR just to not make those classes as Singletons at all.

**Evaluation Criteria**

[2]. All tasks are implemented partially, or two tasks are not implemented at all.

[3]. One task is not implemented at all, or two tasks have major issues.

[4]. One of the tasks does not have a major part of its implementation.

[5]. All tasks are implemented to a full extend.

[5\*]. Typescript decorator has been used to implement Step #4.