Design Patterns Explained
Self-Paced Exercise

The purpose of this document is to provide a hands-on coding experience in implementing design patterns, and to show how patterns can emerge throughout the lifecycle of a software project, as requirements change, grow, and become clearer.

As with all exercises of this type, it is important to balance complexity, which makes the exercise interesting and realistic, with a tight focus on the issues that we are trying to emphasize. Therefore, while the project will be relatively simple, the design ideas brought forth will hopefully be rich, interesting, and empowering.

Here is how this will work:

There is a separate document (DPE_Self-Paced_Exercise_Solutions.pdf) that contains our example solution code**. Do not examine this code ahead of time**, as it will prevent you from engaging with the experience of implementing the patterns yourself.

You will be given a set of requirements in this document. Give yourself ample time to think about the problem, but don't over-design – remember, if you keep the code qualities, principles, and practices you learned in the course uppermost in mind, then you don't have to anticipate every little thing, and you don't have to overdo your design.

Once you have a solution in place, look at the solution document. There you will find our suggested code solution, with explanations. What we have done will not necessarily be better than what you have done (in fact, we hope not!), but it will probably be useful for you to bring your project into "sync" with the examples shown, so that the exercise will make sense as we move forward.

Once you have reviewed the example solution and made your code look like ours, turn the page again and get another set of requirements. Here again, change your solution to accommodate these new requirements, but don't peek ahead.

As we go, we'll see patterns emerge, so long as we keep code quality high.

Turn the page, and we'll begin.

Step 1: Starting Simply

You are coding up the business objects 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 objects. One will represent the controlling, or Client object that will interact with the GUI, when it is available. The other will represent the object being shipped.

State that will come from the end user via the GUI is:

- ShipmentID (an **`int`** that represents an existing ID, or 0, which means you have to 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 **`double`**, storing the weight of the item in ounces[1])

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.

If you are familiar with unit tests, feel free to write them. Our example was test-driven, in fact. However, unit tests are not required since this is not a course in test-driven development (TDD). That said, considering the *testability* of your design is always a very informing thing to do.

[1] This is good example of the balance of real-vs.-useful examples. Our package has weight, but no dimensions! In the real world it would have both, but our point can be made just using weight, so why make you type in more state variables? The same is true for our single Address field rather than separate city and state fields, etc… We're just trying to keep the focus on patterns and design here.

Things to note:

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

- Since you don't have the actual GUI to call upon for the state, you'll have to dummy that up somewhere. Just make sure it's 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.

Figure 1 illustrates this in the UML.

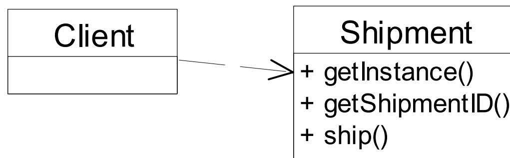

Figure 1: Client using Shipment

STOP HERE AND CODE YOUR SOLUTION. WHEN YOU ARE FINISHED, OPEN THE SOLUTIONS DOCUMENT AND EXAMINE STEP 1.

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 effect 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.

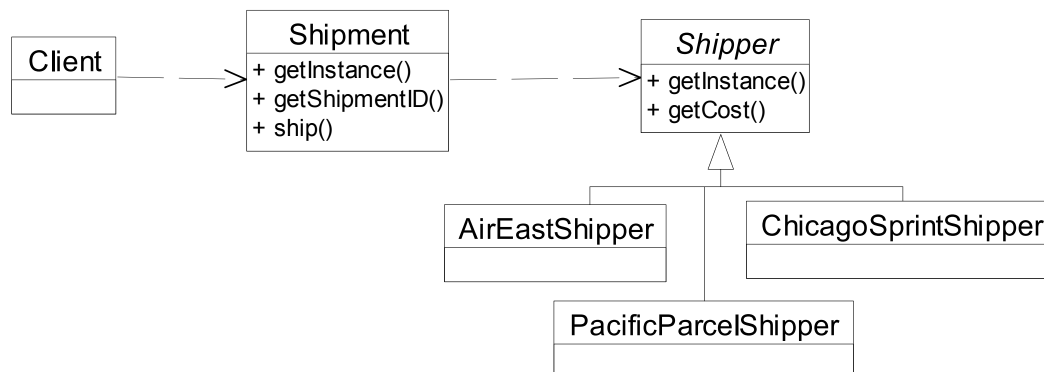

Figure 2: Shipment using Shipper as Strategy

STOP HERE AND CODE YOUR SOLUTION. WHEN YOU ARE FINISHED, OPEN THE SOLUTIONS DOCUMENT AND EXAMINE STEP 2.

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[2].

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:

	Air West	Chicago Sprint	Pacific Parcel
Letter	.39 per ounce	.42 per ounce	.51 per ounce
Package	.25 per ounce	.20 per ounce	.19 per ounce
Oversized	$10 flat in addition to standard package charge	No charge	.02 added per ounce in addition to standard package charge

Figure 3 illustrates this in the UML.

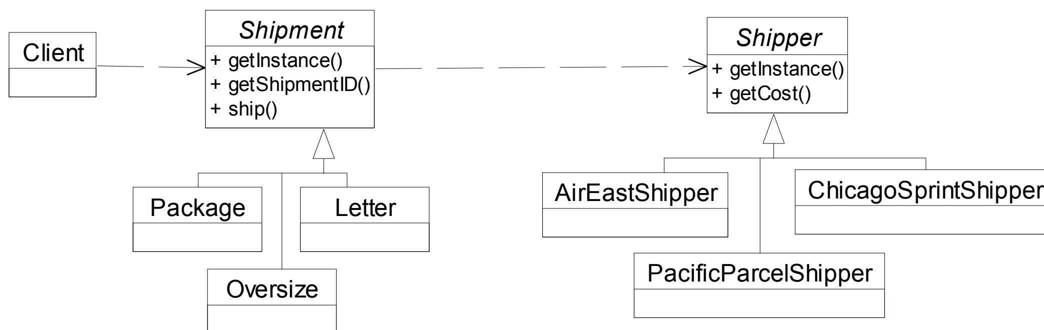

Figure 3: Shipments using Shipper as Bridge

STOP HERE AND CODE YOUR SOLUTION. WHEN YOU ARE FINISHED, OPEN THE SOLUTIONS DOCUMENT AND EXAMINE STEP 3.

[2] …and here again, in the real world this distinction would be based, at least partly, on dimensions rather than weight. We're limiting this to weight only, just to keep the implementations simpler. The ideas are the same, of course.

Copyright © 2005 Net Objectives

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:

Your 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.

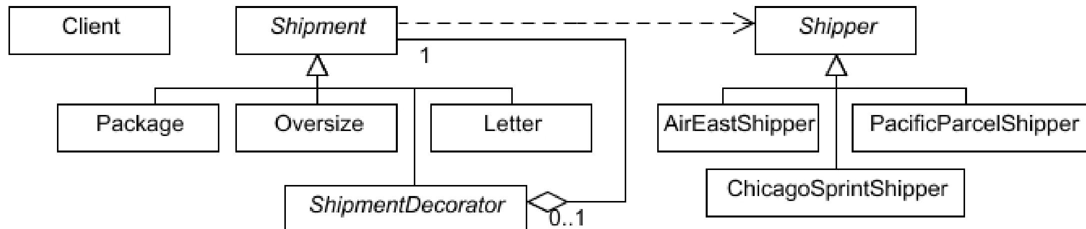

Figure 4: Shipper being decorated.

STOP HERE AND CODE YOUR SOLUTION. WHEN YOU ARE FINISHED, OPEN THE SOLUTIONS DOCUMENT AND EXAMINE STEP 4.