Design Patterns Explained
Self-Paced Exercise Solutions

DO NOT READ THIS DOCUMENT AHEAD OF TIME. THIS IS MEANT TO BE READ AFTER COMPLETING EACH STEP IN THE EXERSIZE. READING IT AHEAD OF TIME WILL DRASTICALLY REDUCE THE VALUE OF THE EXERCISE.

Step 1 Solution 2
Step 2 Solution 6
Step 3 Solution 12
Step 4 Solution 19

STOP! DO NOT TURN THE PAGE UNTIL YOU HAVE ATTEMPTED STEP 1

Copyright © 2005 Net Objectives

Step 1 Solution

Client.cs

```
using System;
using PackageHandler;

namespace PackageHandler
{
    class Client
    {
            private int ShipmentID;
            private string toAddress;
            private string fromAddress;
            private string toZipCode;
            private string fromZipCode;
            private Shipment myShipment;
            private double weight;

            static void Main(string[] args)
            {
                  Client.getInstance().executeShipment();
            }

            private Client(){}
            public static Client getInstance(){ return new Client();}

            public void executeShipment()
            {
                  getShipmentDetails();
                  myShipment = Shipment.getInstance(ShipmentID, toAddress, fromAddress, toZipCode,
                                    fromZipCode, weight);
                  Console.WriteLine(myShipment.ship());
            }
```

```
This is encapsulation of construction.
We always do this at the minimum.
```

```
Programming by intention means that the
"use of the GUI" is kept in one place,
making it easier to change later
```

```
Using the encapsulated constructor of
Shipment (see Shipment.cs below)
```

```
            private void getShipmentDetails()
            {
                  //TODO: Change to use GUI directly,
                  //       or use the Mediator Pattern
                  ShipmentID = 17263;
                  toAddress = "1313 Mockingbird Lane, Tulsa, OK";
                  toZipCode = "67721";
                  fromAddress = "12292 4th Ave SE, Bellevue, Wa";
                  fromZipCode = "92021";
                  weight = 10.00;
            }
      }
}
```

Stubbed out for now, so we can run our code. When the GUI gets done, we'll re-code this… or, we could use another object to represent the GUI, which we'll show in a later step

Shipment.cs

```
using System;
namespace PackageHandler
{
    public class Shipment
    {
            private int myShipmentID;
            private string myToAddress;
            private string myFromAddress;
            private string myToZipCode;
            private string myFromZipCode;
            private double myWeight;

            private Shipment(int shipmentID, string toAddress, string fromAddress, string toZipCode,
                        string fromZipCode, double weight)
            {
                  // allow for specifying shipmentID on entry
                  if (shipmentID == 0)
                  {
                        shipmentID = getShipmentID();
                  }
                  this.myShipmentID = shipmentID;
```

Programming by intention puts this behavior in a separate method.

```
        this.myToAddress = toAddress;
        this.myFromAddress = fromAddress;
        this.myToZipCode = toZipCode;
        this.myFromZipCode = fromZipCode;
        this.myWeight = weight;
}

public static Shipment getInstance(int shipmentID, string toAddress, string fromAddress,
                string toZipCode, string fromZipCode, double weight)
{
        return new Shipment(shipmentID, toAddress, fromAddress, toZipCode, fromZipCode, weight);
}

virtual public int ShipmentID
{
        get {return myShipmentID;}
}

virtual public string ToAddress
{
        get{ return myToAddress;}
        set{ myToAddress = value;}
}

virtual public string FromZipCode
{
        get {return myFromZipCode;}
        set {myFromZipCode = value;}
}

virtual public string ToZipCode
{
        get {return myToZipCode;}
        set {myToZipCode = value;}
}

virtual public string FromAddress
{
```

This is encapsulation of construction again.

Copyright © 2005 Net Objectives

```
            get {return myFromAddress;}
            set {myFromAddress = value;}
        }

        private static int lastID= 0;
        private int getShipmentID()
        {
            //TODO: Implement obtaining next unique ID from ID manager
            return ++lastID;
        }

        virtual protected double calculateCost(double weight)
        {
            return weight * .39;
        }

        virtual public string ship()
        {
            double cost = calculateCost(this.myWeight);

            string response;
            response = "Your Shipment with the ID " + this.myShipmentID;
            response += "\nwill be picked up from " + this.myFromAddress + " " + this.myFromZipCode;
            response += "\nand shipped to " + this.myToAddress + " " + this.myToZipCode;
            response += "\nCost = " + cost;
            return response;
        }
    }
}
```

Here again, the "stubbed out" mechanism for generating the ID when needed is put in its own method, so that when the persistence layer is ready, and we can use the "real" mechanism to get a truly unique ID, we'll have one place to make the change to our code

Cost algorithm

Rather than putting the cost algorithm here, we program by intention and put it in its own method.

STOP! DO NOT TURN THE PAGE UNTIL YOU HAVE ATTEMPTED STEP 2

Copyright © 2005 Net Objectives

Step 2 Solution

(Changes to existing code, and entirely new code will be shown in **boldface font** from this point forward)

Client.cs

```
using System;
using PackageHandler;

namespace PackageHandler
{
    class Client
    {
            private int ShipmentID;
            private string toAddress;
            private string fromAddress;
            private string toZipCode;
            private string fromZipCode;
            private Shipment myShipment;
            private double weight;

            static void Main(string[] args)
            {
                  Client.getInstance().executeShipment();
            }

            private Client(){}
            public static Client getInstance(){ return new Client();}

            public void executeShipment()
            {
                  getShipmentDetails();
                  myShipment = Shipment.getInstance(ShipmentID, toAddress, fromAddress,
                              toZipCode, fromZipCode, weight);
                  Console.WriteLine(myShipment.ship(Shipper.getInstance(fromZipCode)));
            }
```

We're going to "grow" a Strategy Pattern here to handle the varying cost algorithms. In the tradition Gang of Four Strategy, the client will "hand in" the strategy to use. Note we have only one small change in one place here, and also note that we're using an encapsulated constructor once again,

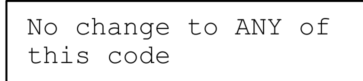
```
            private void getShipmentDetails()
            {
                  //TODO: Change to use GUI directly, or use the Mediator Pattern
                  ShipmentID = 17263;
                  toAddress = "1313 Mockingbird Lane, Tulsa, OK";
                  toZipCode = "67721";
                  fromAddress = "12292 4th Ave SE, Bellevue, Wa";
                  fromZipCode = "92021";
                  weight = 10.00;
            }
      }
}
```

Shipment.cs

```
using System;

namespace PackageHandler
{
    public class Shipment
    {
        private int myShipmentID;
            private string myToAddress;
            private string myFromAddress;
            private string myToZipCode;
            private string myFromZipCode;
            private double myWeight;

            private Shipment(int shipmentID, string toAddress, string fromAddress, string toZipCode,
                             string fromZipCode, double weight)
            {
                  // allow for specifying shipmentID on entry
                  if (shipmentID == 0)
                  {
                        shipmentID = getShipmentID();
                  }
```

No Changes

```
        this.myShipmentID = shipmentID;
        this.myToAddress = toAddress;
        this.myFromAddress = fromAddress;
        this.myToZipCode = toZipCode;
        this.myFromZipCode = fromZipCode;
        this.myWeight = weight;
}

public static Shipment getInstance(int shipmentID, string toAddress, string fromAddress,
            string toZipCode, string fromZipCode, double weight)
{
        return new Shipment(shipmentID, toAddress, fromAddress, toZipCode, fromZipCode, weight);
}


virtual public int ShipmentID
{
        get {return myShipmentID;}
}

virtual public string ToAddress
{
        get{ return myToAddress;}
        set{ myToAddress = value;}
}

virtual public string FromZipCode
{
        get {return myFromZipCode;}
        set {myFromZipCode = value;}
}


virtual public string ToZipCode
{
        get {return myToZipCode;}
        set {myToZipCode = value;}
}
```



```
        virtual public string FromAddress
        {
              get {return myFromAddress;}
              set {myFromAddress = value;}
        }

        private static int lastID= 0;
        private int getShipmentID()
        {
              //TODO: Implement obtaining next unique ID from ID manager
              return ++lastID;
        }

        virtual protected double calculateCost(Shipper ShipperToUse, double weight)
        {
              return ShipperToUse.getCost(weight);
        }

        virtual public string ship(Shipper ShipperToUse)
        {
              double cost = calculateCost(ShipperToUse, this.myWeight);

              string response;
              response = "Your Shipment with the ID " + this.myShipmentID;
              response += "\nwill be picked up from " + this.myFromAddress + " " + this.myFromZipCode;
              response += "\nand shipped to " + this.myToAddress + " " + this.myToZipCode;
              response += "\nCost = " + cost;
              return response;
        }
    }
}
```

This is a design change from implementing a simple algorithm to delegating to a Strategy object. Note that we've added a single parameter, and changed one line of code – Programming by Intention again!

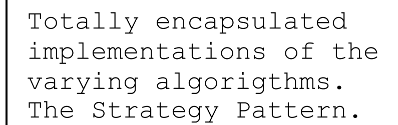
Shipper.cs

```
using System;
namespace PackageHandler
{
      public abstract class Shipper
      {
            public static Shipper getInstance(String fromZipCode)
            {
                  Shipper rval;
                  switch(fromZipCode[0])
                  {
                        case('1'):
                        case('2'):
                        case('3'):
                              rval = new AirEastShipper();
                              break;
                        case('4'):
                        case('5'):
                        case('6'):
                              rval = new ChicagoSprintShipper();
                              break;
                        case('7'):
                        case('8'):
                        case('9'):
                              return new PacificParcelShipper();
                        default:
                              rval = new AirEastShipper();
                              break;
                  }
                  return rval;
            }

            public abstract double getCost(double weight);
      }
```

New abstraction, new service. The encapsulated constructor hides this variation from all other entities in the system. The mapping of certain zip codes to certain shippers is an encapsulated rule, and kept here in once place (and therefore is easy to change later).

An option would be to use a ShipperFactory to build the right Shipper, and this would also encapsulate the construction of the Shippers and the zip code rules. If you did that, you did good. However if you think that's overkill now, remember you can "grow" the factory later, and delegate to it from here at that time

```
public class AirEastShipper : Shipper
	{
		private readonly double AIREAST_RATE = .39;
		public override double getCost(double weight)
		{
			return weight * AIREAST_RATE;
		}
	}

	public class ChicagoSprintShipper : Shipper
	{
		private readonly double CHICAGOSPRINT_RATE = .42;
		public override double getCost(double weight)
		{
			return weight * CHICAGOSPRINT_RATE;
		}
	}

	public class PacificParcelShipper : Shipper
	{
		private readonly double PACIFICPARCEL_RATE = .51;
		public override double getCost(double weight)
		{
			return weight * PACIFICPARCEL_RATE;
		}
	}
}
```



STOP! DO NOT TURN THE PAGE UNTIL YOU HAVE ATTEMPTED STEP 3

Copyright © 2005 Net Objectives

Step 3 Solution

Client.cs – No change at all!

Shipment.cs

```
using System;

namespace PackageHandler
{
    public abstract class Shipment
    {
        private int myShipmentID;
            private string myToAddress;
            private string myFromAddress;
            private string myToZipCode;
            private string myFromZipCode;
            private double myWeight;

            public static readonly double MAX_WEIGHT_LETTER_OZ = 15;
            public static readonly double MAX_WEIGHT_PACKAGE_OZ = 160;

            protected Shipment(int shipmentID, string toAddress, string fromAddress,
                               string toZipCode, string fromZipCode, double weight)
            {
                  // allow for specifying shipmentID on entry
                  if (shipmentID == 0)
                  {
                        shipmentID = getShipmentID();
                  }
                  this.myShipmentID = shipmentID;
                  this.myToAddress = toAddress;
                  this.myFromAddress = fromAddress;
                  this.myToZipCode = toZipCode;
                  this.myFromZipCode = fromZipCode;
```

Design change! And note no change to Client. Encapsulated Construction pays off again.

Needed constants are added here.

Copyright © 2005 Net Objectives

```
        this.myWeight = weight;
}

public static Shipment getInstance(int shipmentID,
            string toAddress, string fromAddress,
            string toZipCode, string fromZipCode, double weight)
{
      if (weight > MAX_WEIGHT_PACKAGE_OZ)
      {
            return new Oversized(shipmentID, toAddress, fromAddress,
                        toZipCode, fromZipCode, weight);
      }
      else if (weight > MAX_WEIGHT_LETTER_OZ)
      {
            return new Package(shipmentID, toAddress, fromAddress,
                        toZipCode, fromZipCode, weight);
      }
      else
      {
            return new Letter(shipmentID, toAddress, fromAddress,
                        toZipCode, fromZipCode, weight);
      }
}


virtual public int ShipmentID
{
      get {return myShipmentID;}
}

virtual public string ToAddress
{
      get{ return myToAddress;}
      set{ myToAddress = value;}
}

virtual public string FromZipCode
{
```

The only other change to this class is to the encapsulated constructor. Using the weight parameter, which was already part of this method, the rule binding certain weights to certain Shipment types is, again, encapsulated here. And as before, when the Shipper grew into a Strategy, we could use a separate factory now, or later when it becomes necessary.

Copyright © 2005 Net Objectives

```
        get {return myFromZipCode;}
        set {myFromZipCode = value;}
    }


    virtual public string ToZipCode
    {
        get {return myToZipCode;}
        set {myToZipCode = value;}
    }

    virtual public string FromAddress
    {
        get {return myFromAddress;}
        set {myFromAddress = value;}
    }

    private static int lastID= 0;
    private int getShipmentID()
    {
        //TODO: Implement obtaining next unique ID from ID manager
        return ++lastID;
    }

    protected abstract double calculateCost(Shipper ShipperToUse, double weight); <-

    virtual public string ship(Shipper ShipperToUse)
    {
        double cost = calculateCost(ShipperToUse, this.myWeight);

        string response;
        response = "Your Shipment with the ID " + this.myShipmentID;
        response += "\nwill be picked up from " + this.myFromAddress + " " + this.myFromZipCode;
        response += "\nand shipped to " + this.myToAddress + " " + this.myToZipCode;
        response += "\nCost = " + cost;
        return response;
    }
}
```

Design change – delegating implementation to subclasses, aka The Template Method.

```
public class Letter : Shipment
{
      public Letter(int shipmentID, string toAddress, string fromAddress,
                  string toZipCode, string fromZipCode, double weight):
                  base(shipmentID, toAddress, fromAddress, toZipCode,
                  fromZipCode, weight)          {}

      protected override double calculateCost(Shipper ShipperToUse, double weight)
      {
            return ShipperToUse.getLetterCost(weight);
      }
}

public class Package : Shipment
{
      public Package(int shipmentID, string toAddress, string fromAddress,
                  string toZipCode, string fromZipCode, double weight):
      base(shipmentID, toAddress, fromAddress, toZipCode,
                  fromZipCode, weight){}
      protected override double calculateCost(Shipper ShipperToUse, double weight)
      {
            return ShipperToUse.getPackageCost(weight);
      }
}

public class Oversized : Shipment
{
      public Oversized(int shipmentID, string toAddress, string fromAddress,
                  string toZipCode, string fromZipCode, double weight):
      base(shipmentID, toAddress, fromAddress, toZipCode,
                  fromZipCode, weight){}
      protected override double calculateCost(Shipper ShipperToUse, double weight)
      {
            return ShipperToUse.getPackageCost(weight)+ShipperToUse.getOversizeSurcharge(weight);
      }
}
}
```

These three classes represent that Shipment, which was concrete, is now an abstraction. Note that each one uses the Shipper service in a different way (Letter calls getLetterCost() while Oversized calls both getPackageCost() and getOversizeSurcharge(), etc...)

The different Shippers (below) have grown a larger interface, and thus our Strategy has grown into a Bridge.

This is not at all uncommon, and illustrates the nature of software to evolve new designs over time. This is why we look for practices that make this easier to do.

Copyright © 2005 Net Objectives

Shipper.cs

```
using System;

namespace PackageHandler
{
	public abstract class Shipper
	{
		public static Shipper getInstance(String fromZipCode)
		{
			Shipper rval;
			switch(fromZipCode[0])
			{
				case('1'):
				case('2'):
				case('3'):
					rval = new AirEastShipper();
					break;
				case('4'):
				case('5'):
				case('6'):
					rval = new ChicagoSprintShipper();
					break;
				case('7'):
				case('8'):
				case('9'):
					return new PacificParcelShipper();
				default:
					rval = new AirEastShipper();
					break;
			}
			return rval;
		}

		public abstract double getLetterCost(double weight);
		public abstract double getPackageCost(double weight);
		public abstract double getOversizeSurcharge(double weight);
```

The interface of our Shipper implementations changes because our simple Strategy has evolved into a Bridge

```
}

public class AirEastShipper : Shipper
{
	private readonly double AIREAST_LETTER_RATE = .39;
	private readonly double AIREAST_PACKAGE_RATE = .24;

	public override double getLetterCost(double weight)
	{
		return weight * AIREAST_LETTER_RATE;
	}

	public override double getPackageCost(double weight)
	{
			return weight * AIREAST_PACKAGE_RATE;
	}

	public override double getOversizeSurcharge(double weight)
	{
		return 10.00;
	}
}

public class ChicagoSprintShipper : Shipper
{
	private readonly double CHICAGOSPRINT_LETTER_RATE = .42;
	private readonly double CHICAGOSPRINT_PACKAGE_RATE = .20;

	public override double getLetterCost(double weight)
	{
		return weight * CHICAGOSPRINT_LETTER_RATE;
	}

	public override double getPackageCost(double weight)
	{
			return weight * CHICAGOSPRINT_PACKAGE_RATE;
	}
```

These entities are the same, but these implementations are different. Note that the encapsulation we put in place in the beginning (by using the Strategy pattern) means that we are making these changes in an encapsulated place, and thus there is far less danger of introducing bugs.

```
		public override double getOversizeSurcharge(double weight)
		{
			return 0.00;
		}
	}

	public class PacificParcelShipper : Shipper
	{
		private readonly double PACIFICPARCEL_LETTER_RATE = .51;
		private readonly double PACIFICPARCEL_PACKAGE_RATE = .19;
		private readonly double PACIFICPARCEL_OVERSIZE_SURCHARGE_RATE = .02;

		public override double getLetterCost(double weight)
		{
			return weight * PACIFICPARCEL_LETTER_RATE;
		}

		public override double getPackageCost(double weight)
		{
				return weight * PACIFICPARCEL_PACKAGE_RATE;
		}

		public override double getOversizeSurcharge(double weight)
		{
			return weight * PACIFICPARCEL_OVERSIZE_SURCHARGE_RATE;
		}
	}
}
```

STOP! DO NOT TURN THE PAGE UNTIL YOU HAVE ATTEMPTED STEP 4

Step 4 Solution

Client.cs

```
using System;
using PackageHandler;
namespace PackageHandler
{
    class Client
    {
            private PackageHandlerMediator myMediator =
                  PackageHandlerMediator.getInstance();
            private int ShipmentID;
            private string toAddress;
            private string fromAddress;
            private string toZipCode;
            private string fromZipCode;
            private Shipment myShipment;
            private double weight;

            static void Main(string[] args)
            {
                  Client.getInstance().executeShipment();
            }

            private Client(){}
            public static Client getInstance(){ return new Client();}

            public void executeShipment()
            {
                  getShipmentDetails();
                  myShipment = Shipment.getInstance(ShipmentID, toAddress,
                        fromAddress, toZipCode, fromZipCode, weight);
                  Console.WriteLine(myShipment.ship(Shipper.getInstance(fromZipCode)));
```

We're going to delegate the "GUI Issue" to another object because, as we'll see, there are now *two* entities that have to interact with it, and yet we still don't have the actual GUI. This entity, which is a kind of Mediator Pattern (or may become one) eliminates this redundancy. Note that it is implemented as a Singleton, which makes it easy for us to "get the instance" here and also encapsulates construction, which we know is a very good thing.

```
        }

        private void getShipmentDetails()
        {
            ShipmentID = this.myMediator.getShipmentID();
            toAddress = this.myMediator.getToAddress();
            toZipCode = this.myMediator.getToZipCode();
            fromAddress = this.myMediator.getFromAddress();
            fromZipCode = this.myMediator.getFromZipCode();
            weight = this.myMediator.getWeight();
        }
    }
}
```

Our "Stubbed behavior" has simply been moved to another object. Note very little, limited change is needed to make this happen.

PackageHandlerMediator.cs

```
using System;
namespace PackageHandler
{
    public class PackageHandlerMediator
    {
        // TODO: This class is stubbed out, will be wired to the GUI when it exists
        private static PackageHandlerMediator _instance = new PackageHandlerMediator();
        private PackageHandlerMediator(){}

        public static PackageHandlerMediator getInstance(){return _instance;}
        public int getShipmentID(){ return 17263;}
        public string getToAddress(){ return "1313 Mockingbird Lane, Tulsa, OK";}
        public string getToZipCode(){ return "67721";}
        public string getFromAddress(){ return "12292 4th Ave SE, Bellevue, Wa";}
        public string getFromZipCode(){ return "92021";}
        public double getWeight(){ return 10.00;}

        //Added to Mediator to support decorating the package
        public bool markFragile(){ return false;}
        public bool markDoNotLeave() {return false;}
        public bool markReturnReceiptRequested() {return false;}
```

Singleton

Stubbed behavior moved from Client.

New stubs needed for the Decorator (see below)

Copyright © 2005 Net Objectives

```
}}
```

Copyright © 2005 Net Objectives

```
using System;

namespace PackageHandler
{
    public abstract class Shipment
    {
        private int myShipmentID;
            private string myToAddress;
            private string myFromAddress;
            private string myToZipCode;
            private string myFromZipCode;
            private double myWeight;

            public static readonly double MAX_WEIGHT_LETTER_OZ = 15;
            public static readonly double MAX_WEIGHT_PACKAGE_OZ = 160;

            protected Shipment(){}

            protected Shipment(int shipmentID, string toAddress, string fromAddress,
                  string toZipCode, string fromZipCode, double weight)
            {
                  if (shipmentID == 0)
                  {
                        shipmentID = getShipmentID();
                  }
                  this.myShipmentID = shipmentID;
                  this.myToAddress = toAddress;
                  this.myFromAddress = fromAddress;
                  this.myToZipCode = toZipCode;
                  this.myFromZipCode = fromZipCode;
                  this.myWeight = weight;
            }

            public static Shipment getInstance(int shipmentID, string toAddress,
                  string fromAddress, string toZipCode, string fromZipCode, double weight)
            {
                  return ShipmentFactory.getInstance().getShipment(shipmentID, toAddress,
                  fromAddress, toZipCode, fromZipCode, weight);
```

A factory is now definitely needed because we're going to add decorators to the Shipments. Note how small this change is…

```
}

virtual public int ShipmentID
{
      get {return myShipmentID;}
}

virtual public string ToAddress
{
      get{ return myToAddress;}
      set{ myToAddress = value;}
}

virtual public string FromZipCode
{
      get {return myFromZipCode;}
      set {myFromZipCode = value;}
}


virtual public string ToZipCode
{
      get {return myToZipCode;}
      set {myToZipCode = value;}
}

virtual public string FromAddress
{
      get {return myFromAddress;}
      set {myFromAddress = value;}
}

private static int lastID= 0;
private int getShipmentID()
{
      //TODO: Implement obtaining next unique ID from ID manager
      return ++lastID;
```

No Changes

```
	}

	protected abstract double calculateCost(Shipper ShipperToUse, double weight);

	virtual public string ship(Shipper ShipperToUse)
	{
		double cost = calculateCost(ShipperToUse, this.myWeight);

		string response;
		response = "Your Shipment with the ID " + this.myShipmentID;
		response += "\nwill be picked up from " + this.myFromAddress + " " + this.myFromZipCode;
		response += "\nand shipped to " + this.myToAddress + " " + this.myToZipCode;
		response += "\nCost = " + cost;
		return response;
	}
}
```

No Changes

```
public class Letter : Shipment
{
	public Letter(int shipmentID, string toAddress, string fromAddress,
		string toZipCode, string fromZipCode, double weight):
		base(shipmentID, toAddress, fromAddress, toZipCode, fromZipCode, weight)
	{}

	protected override double calculateCost(Shipper ShipperToUse, double weight)
	{
		return ShipperToUse.getLetterCost(weight);
	}
}

public class Package : Shipment
{
	public Package(int shipmentID, string toAddress, string fromAddress,
		string toZipCode, string fromZipCode, double weight):
	base(shipmentID, toAddress, fromAddress, toZipCode, fromZipCode, weight)
	{}
	protected override double calculateCost(Shipper ShipperToUse, double weight)
	{
```

Copyright © 2005 Net Objectives

```
                return ShipperToUse.getPackageCost(weight);
            }
        }

        public class Oversized : Shipment
        {
            public Oversized(int shipmentID, string toAddress, string fromAddress,
                string toZipCode, string fromZipCode, double weight):
            base(shipmentID, toAddress, fromAddress, toZipCode, fromZipCode, weight)
            {}
            protected override double calculateCost(Shipper ShipperToUse, double weight)
            {
                return ShipperToUse.getPackageCost(weight)+ShipperToUse.getOversizeSurcharge(weight);
            }
        }
    }
```

ShipmentFactory.cs

```
namespace PackageHandler
{
    public class ShipmentFactory
    {
        private PackageHandlerMediator myMediator = PackageHandlerMediator.getInstance();

        private static ShipmentFactory _instance = new ShipmentFactory();
        private ShipmentFactory(){}
        public static ShipmentFactory getInstance(){ return _instance;}

        public Shipment getShipment(int shipmentID, string toAddress,
            string fromAddress, string toZipCode, string fromZipCode, double weight)
        {
            Shipment basePackage = getBasePackage(shipmentID, toAddress, fromAddress,
                                        toZipCode, fromZipCode, weight);
            Shipment decoratedPackage = decoratePackage(basePackage);
            return decoratedPackage;
        }
```

```
    private Shipment getBasePackage(int shipmentID, string toAddress,
          string fromAddress, string toZipCode, string fromZipCode, double weight)
    {
          if (weight > Shipment.MAX_WEIGHT_PACKAGE_OZ)
          {
                return new Oversized(shipmentID, toAddress, fromAddress,
                      toZipCode, fromZipCode, weight);
          }
          else if (weight > Shipment.MAX_WEIGHT_LETTER_OZ)
          {
                return new Package(shipmentID, toAddress, fromAddress,
                      toZipCode, fromZipCode, weight);
          }
          else
          {
                return new Letter(shipmentID, toAddress, fromAddress,
                      toZipCode, fromZipCode, weight);
          }
    }

    private Shipment decoratePackage(Shipment basePackage)
    {
          Shipment returnPackage = basePackage;
          if(myMediator.markFragile()) returnPackage =
                new FragileDecorator(returnPackage);
          if(myMediator.markDoNotLeave()) returnPackage =
                new DoNotLeaveDecorator(returnPackage);
          if(myMediator.markReturnReceiptRequested()) returnPackage =
                new ReturnReceiptRequestedDecorator(returnPackage);
          return returnPackage;
    }

  }
}
```

This behavior was in the getInstance() method of Shipment, but has moved here as-is, now that we have a full-fledged factory.

This is the new "decorating" behavior. Note we are adding this to the base behavior, without having to change the base behavior.

Note, also, the use of the GUI stub, or "mediator" here.

ShipmentDecorator.cs

```
using System;

namespace PackageHandler
{
	public abstract class ShipmentDecorator : Shipment
	{
		private Shipment nextShipment;
		public ShipmentDecorator(Shipment nextShipment)
		{
			this.nextShipment = nextShipment;
		}

		protected override double calculateCost(Shipper ShipperToUse, double weight)
		{
			return 0;
		}


		public override string ship(Shipper shipperToUse)
		{
			return nextShipment.ship(shipperToUse);
		}
	}

	public class FragileDecorator : ShipmentDecorator
	{
		public FragileDecorator(Shipment nextShipment):base(nextShipment){}
		public override string ship(Shipper shipperToUse)
		{
			string rval = base.ship(shipperToUse);
			return rval + "\n**MARK FRAGILE**";
		}
	}

	public class DoNotLeaveDecorator : ShipmentDecorator
```

These classes, all new, are an implementation of the Decorator Pattern. Note, again, that we have mostly "added stuff" rather than "changing stuff", and thus are following the Open-Closed Principle to a great degree.

Copyright © 2005 Net Objectives

```
	{
		public DoNotLeaveDecorator(Shipment nextShipment):base(nextShipment){}
		public override string ship(Shipper shipperToUse)
		{
			string rval = base.ship(shipperToUse);
			return rval + "\n**MARK DO NOT LEAVE IF ADDRESS NOT AT HOME**";
		}
	}

	public class ReturnReceiptRequestedDecorator : ShipmentDecorator
	{
		public ReturnReceiptRequestedDecorator(Shipment nextShipment):base(nextShipment){}
		public override string ship(Shipper shipperToUse)
		{
			string rval = base.ship(shipperToUse);
			return rval + "\n**MARK RETURN RECEIPT REQUESTED**";
		}
	}

}
```