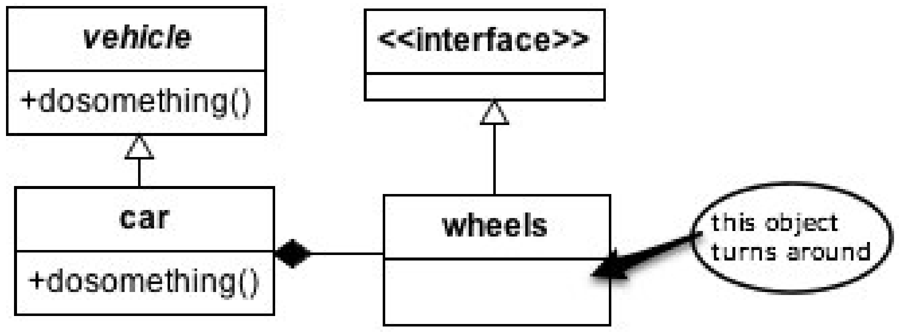
Name______Alexander Toenniessen________
150 points possible (10 of which are for identifying where answers came from)

Rename the file to include your name (e.g. capaultmidterm). Open book, open notes (including our website and the links on it – no Google/Bing searches). NOTE: It is NOT permissible to use a midterm exam for notes from someone you know who took this course previously. Discuss your answers with no one until after the due date. You are on the honor system with regards to these items. You agreed to a code of ethics document as part of your acceptance into this department. Honor it!
IMPORTANT NOTE 1: For each problem, describe where your answer came from (self, book, notes, web address). Failure to do so will result in a loss of 10 points.

IMPORTANT NOTE2: NO LATE EXAMS WILL BE ACCEPTED FOR POINTS. CANVAS WILL DISABLE TURN-IN. YOU HAVE 5 DAYS TO COMPLETE THE EXAM – THERE ARE NO EXCUSES FOR A LATE EXAM.

For most questions you **may** use words, class diagrams or Java/C# code to illustrate your answer, unless the question asks for a specific format. When you draw a class diagram, indicate the **methods** that make the pattern work. Also specify the type of **relation** (inheritance / composition) between classes and **type** of class (interface / abstract class / regular) as indicated in the sample UML class diagram on the right. You may also use the terms "is-a" or "has-a" to indicate the relationships. For each class & interface in your diagram indicate what it does. Use a circle to describe the **responsibilities** of each class, as necessary. The more detail you include; the better chance you have at earning full credit.



1. **(8 points)** Describe what the design principle "Tell, don't ask" is about. Include in your discussion specifically how each of the Pillars of OO help achieve this principle.

(found in lecture notes, my mind and assumptions, and this link http://penguin.ewu.edu/cscd454/notes/OO_principles.pdf)

The design principle of "Tell, don't ask" is programming with the intention of good encapsulation in mind. All objects and their data should be told what to do (method calls), not asked for their data to be manipulated.

- Abstraction allows for multiple versions of a concrete implementation of the object being told what to do to exist. This allows for an object with varying data types unbeknownst to the client to be used for the same operations, such as an area method for a circle and square. The client doesn't care how to calculate the area; they just want to know it.

- Encapsulation is the whole goal of "Tell, don't ask" as the client should never be able to access data that is uniquely available to the underlying object. All the client needs to be able to do is invoke methods that manipulate the data of an object to produce their required answers. Encapsulation heavily uses abstraction, as it allows the use of an interface to separate the client from the workings of the program that they don't really care about anyways and prevents accidental tampering of a programs inner workings.
- Polymorphism allows for even greater generalizations of an object and more functionality. An example of a shape, whether it be a circle, square, triangle or something else. Depending on what measurements a user enters, a different shape is formed. The user can call for the area to be measured and each shape knows how to do just that. Polymorphism also allows for runtime changes in what instance of a shape is being used, making a program even more versatile to the user without their knowledge of the complexity of the structure
- Inheritance is a general violator of the rule of encapsulation, but still allows for telling, not asking. A parent class tells a child class what it can do and the child class allows for extension on the parent's functionality by creation of new concrete methods.

2. (**6 points**) Why does inheritance violate encapsulation principles?

(my brain and the above answers)

Inheritance violates encapsulation principles because the child knows everything about the parent class, including data, which can be modified by the child. With proper encapsulation, only the class the data is in should be able to modify or work with the data.

3. **(10 points)** How are the Adapter and Façade patterns similar? How do they differ? Be **<u>very</u>** specific to ensure full credit.

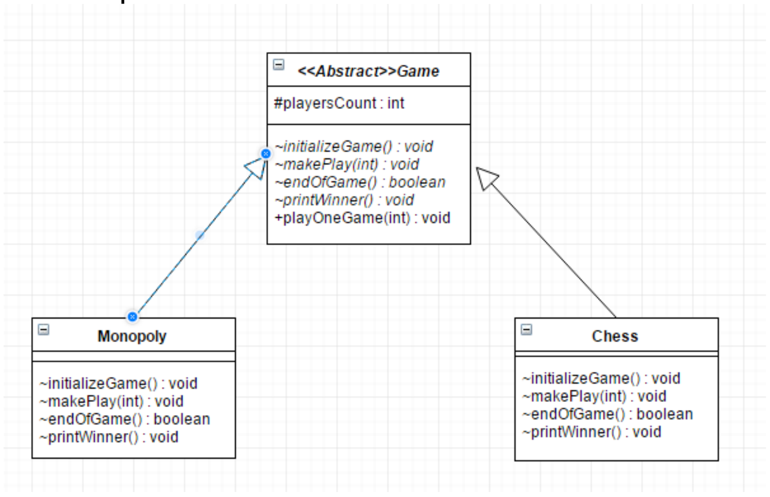
Source: http://penguin.ewu.edu/cscd454/notes/Adapter.pdf

a) Adapter and Façade are similar because they both involve connecting interfaces together and are Structural design patterns.
b) They differ because adapter converts existing interfaces into another that is expected, while façade simplifies the interface to make a complex subsystem easier to use.

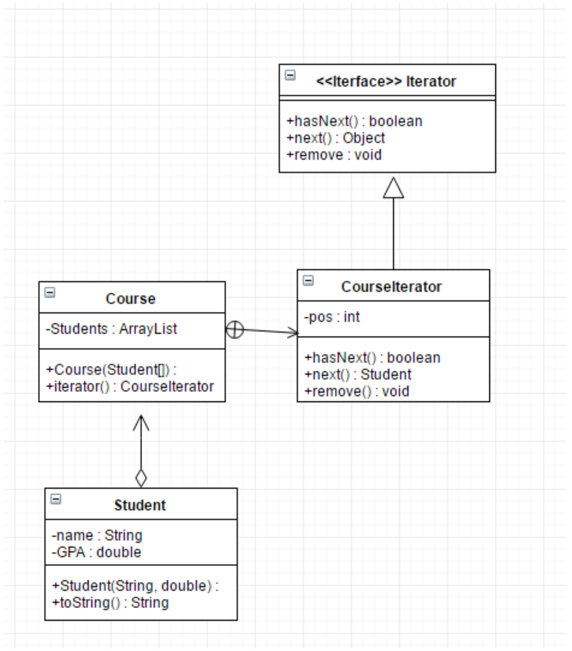
4. (**10 points**) (a) Write a class diagram for the **Template Method *and*** then (b) the **Iterator** design patterns. Note they are separate problems.

Follow the instructions for drawing and describing a class diagram on the first page of this midterm. NOTE: you should have two UML diagrams, one for each pattern. **<u>Each pattern must be applied to a specific scenario – don't give the generic UML for the pattern</u>**. Be sure and include important features that guarantee the correctness of the pattern. The choice of the scenario is up to you. Examples from class are permissible! Once again, be sure and include the important features of each pattern that guarantee the pattern's correctness.

Template Method UML: Sources http://penguin.ewu.edu/cscd454/notes/Template_method.pdf include the basic uml examples and the pseudo code.



Iterator UML: Sources http://penguin.ewu.edu/cscd454/notes/Iterator.pdf and recreating my iterator uml because I realized I missed points on observer since I didn't include the interface implementation as part of the UML. Also based on the UML example description at the top of the exam



5. (**10 points**) (a) Thoroughly describe the following OO principles then (b) list at least one pattern that follows each principle: Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion (SOLID).

Source: http://www.oodesign.com/ and class assignments

1. Single Responsibility: the idea that every method of a class should have one responsibility. If a method does too many operations to achieve one goal, split the other operations up into methods of their own, this greatly improves testing of your code and being able to change code without breaking other aspects of a class. Single Responsibility applies to classes as well, if a class shape does work for circle and square, abstract shape and have a class for just circle operations and just square operations to simplify your code.
    a. A design pattern that follows this principle is factory method which receives information and creates a concrete implementation of a user specified object. Just like how our factory assignment received measurements and a name and was able to form individual classes for Circle, Square, Rectangle and triangle in response. All of those classes had one responsibility, to perform operations on their shapes
2. Open/Closed: The idea of programing with the intent of being open to extensions (forming a new class that inherits the functionality of the super class, extending it past the original classes abilities) and closed to modifications (not having to change code in the class and potentially breaking other aspects of the program, changing everything slightly in the end).
    a. A design pattern that follows this principle is decorator, which extends the functionality of an Object to include other Objects that modify the original in some way. With the Christmas tree assignment, we extended the Christmas tree functionality (i.e., price) to include a variety of decorations that did not modify the underlying structure, just wrapped themselves around it. Thus being open for extension, and closed for modification
3. Liskov Substitution: A class which extends the functionality of a super class must be completely substitutable for the super class (I.e., shape is super class, circle is a shape. Circle must have all of shapes abilities plus its own)
    a. A design pattern that follows this principle is factory method again. The factory creates a shape, the shape can be circle, triangle, etc. All of these different shapes have the same functionality of shape (area) but a triangle could have extended functionality to find the angle given two sides, a circle cannot have that functionality though.
4. Interface Segregation: One all-encompassing interface is not OK! Instead use multiple small interfaces so a client doesn't have functionality they don't need or want to implement.
    a. A design pattern that follows this principle is Observer. instead of having the observer functionality as part of the observable class it was split into two implementations. One that can update all the observers, and one the maintains a list of observers to be updated about the observable object.
5. Dependency Inversion: Taking all that is the same between classes and putting it in an abstraction, so that only methods that differ need implementing between classes of similar functionality.
    a. A design pattern that follows this principle is Factory method AGAIN. In our assignment the shape abstracted variables and methods shared between all shapes, then the concrete versions of the shapes determined how to do it.

6. **(15 points)** List 5 code smells, describe what each means, and describe how to re-factor each.
   Sources: http://penguin.ewu.edu/cscd454/notes/SmellsToRefactorings.pdf and lectures
   1. Deodorant: Writing comments that describe what something is and not why something is. Solve by renaming methods and variables to be intent revealing
   2. Conditional Complexity: using a large if else or switch case to decide what to use in a program. Solve by using strategy design to remove this complexity, having multiple small classes that also will allow for easy runtime changes to what is being used
   3. Duplicate code: when multiple classes have identical or almost identical code. Solve by removing all that is the same between classes to an interface that those classes can then extend.
   4. Freeloader: a class that doesn't do enough to justify having it as a separate class. Solve by putting its functionality in another class.
   5. Inappropriate Intimacy: classes have too much access to each other's underlying data. Solve by using interfaces instead of inheritance if possible or change the visibility of data to deny access.

7. (**6 points**) Describe coupling. Describe cohesion. Which combination of coupling and cohesion is most desirable?

   Source : http://penguin.ewu.edu/cscd454/notes/OOA_D.pdf

   Coupling is the strength of a connection between two classes. Cohesion is when we want a method or class to do just one thing and do it well. The combination of coupling and cohesion most desirable is weak coupling, strong cohesion.

Design Problems

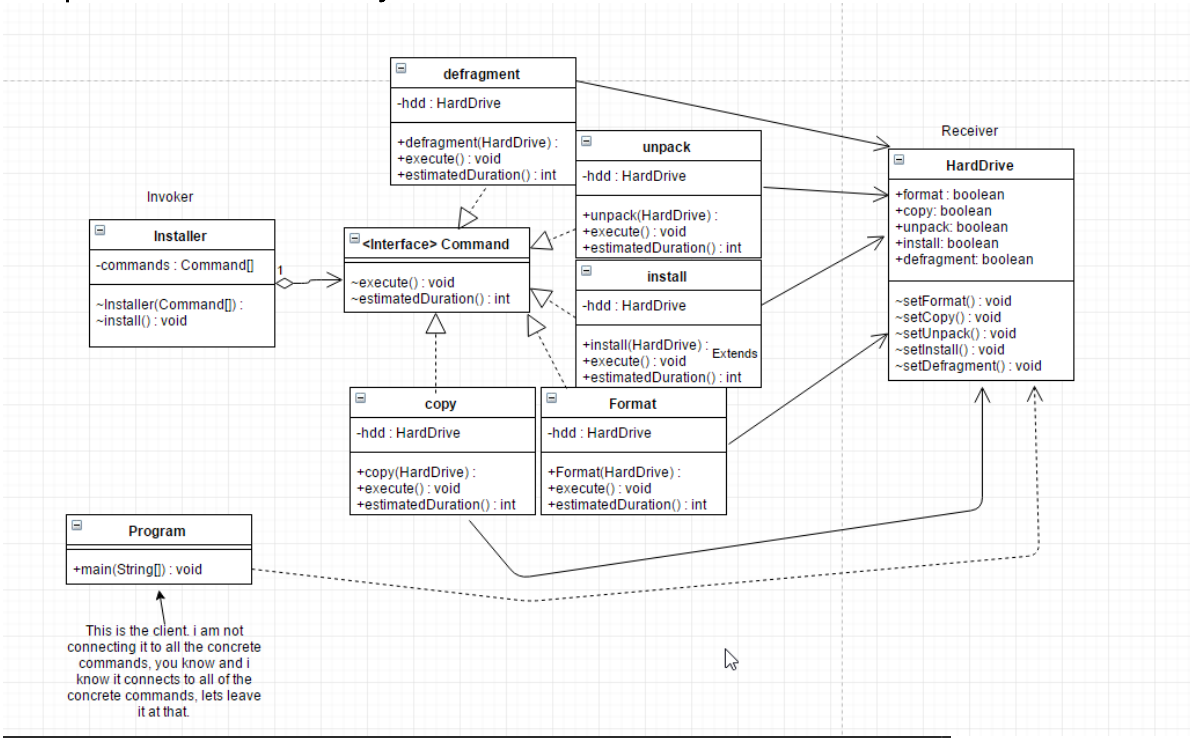
Four design problems are listed on the pages that follow – you must provide solutions for three. If you solve a fourth, that one is worth 10 points extra credit (clearly label which is extra credit if you attempt it, please). Select the most appropriate design pattern to use to address the problem and clearly motivate how it addresses the problem. **Furthermore**, show an appropriate class diagram followed by enough code fragments to illustrate the implementation of your pattern.

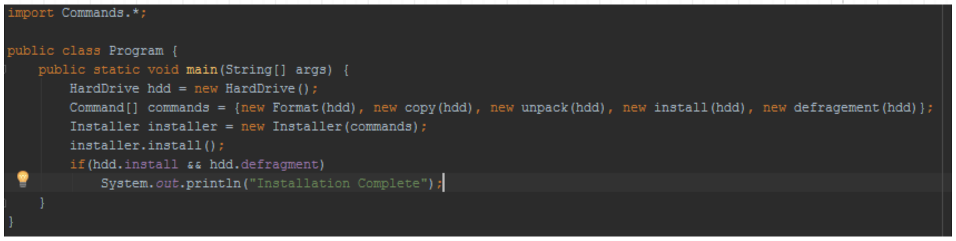
To clarify, you will do **three** things for each problem. **First**, list a pattern that you think best solves the problem along with justification for why you chose it/ why it works. **Second**, draw UML that represents that pattern in the context of the problem. **Third**, include code fragments/snippets that illustrate application of the pattern.

8. **(20 points)**. You must write the installer for a large exotic operating system. This installer executes a number of different tasks that you have to create as well (e.g. format hard drive, defragment, copy files, unpack stuff, install drivers). When your installer executes it calls the different tasks in sequence and it displays a progress bar. Your progress bar must meaningfully reflect how close the program is to completing all the tasks. Tasks are able to estimate themselves how long they will take to complete.

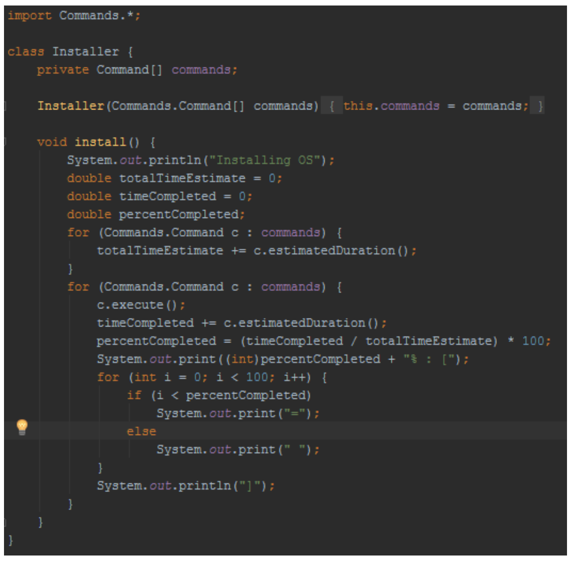
Sources : http://penguin.ewu.edu/cscd454/notes/Command.pdf, Head First into Design Patterns book and A LOT of trial and error

The command design pattern is the pattern meant for this exact type of problem. Each function would reside in its own class, controlled through one interface. This would allow for all of the functionality required of the program available from one control window. The command pattern also can have an estimatedDuration function that can be implemented, providing the ability for a progress bar to reflects how relatively close the program is from completion both individually and as a whole.

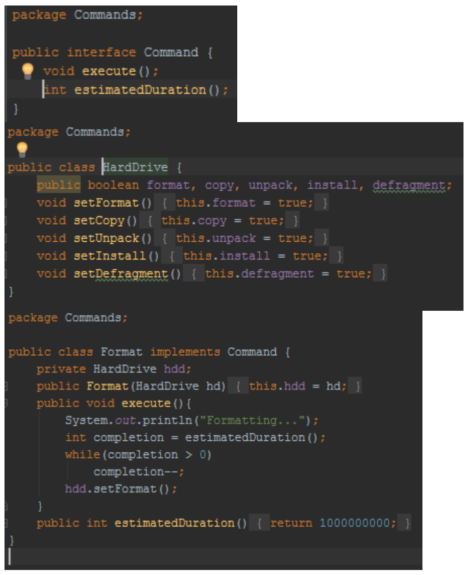




```java
import Commands.*;

public class Program {
    public static void main(String[] args) {
        HardDrive hdd = new HardDrive();
        Command[] commands = {new Format(hdd), new copy(hdd), new unpack(hdd), new install(hdd), new defragement(hdd)};
        Installer installer = new Installer(commands);
        installer.install();
        if(hdd.install && hdd.defragment)
            System.out.println("Installation Complete");
    }
}
```



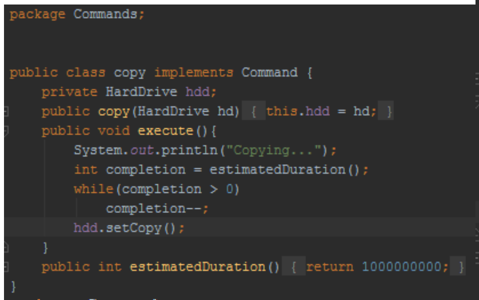
```java
import Commands.*;

class Installer {
    private Command[] commands;

    Installer(Commands.Command[] commands) { this.commands = commands; }

    void install() {
        System.out.println("Installing OS");
        double totalTimeEstimate = 0;
        double timeCompleted = 0;
        double percentCompleted;
        for (Commands.Command c : commands) {
            totalTimeEstimate += c.estimatedDuration();
        }
        for (Commands.Command c : commands) {
            c.execute();
            timeCompleted += c.estimatedDuration();
            percentCompleted = (timeCompleted / totalTimeEstimate) * 100;
            System.out.print((int)percentCompleted + "% : [");
            for (int i = 0; i < 100; i++) {
                if (i < percentCompleted)
                    System.out.print("=");
                else
                    System.out.print(" ");
            }
            System.out.println("]");
        }
    }
}
```



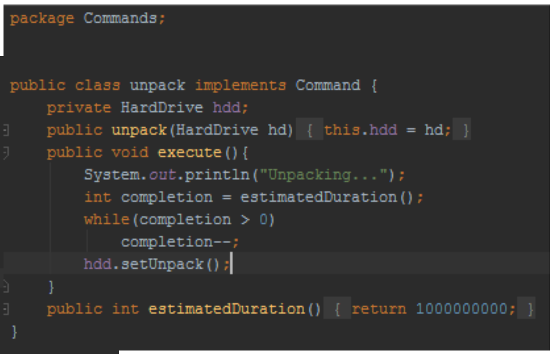
```java
package Commands;

public interface Command {
    void execute();
    int estimatedDuration();
}
```

```java
package Commands;

public class HardDrive {
    public boolean format, copy, unpack, install, defragment;
    void setFormat() { this.format = true; }
    void setCopy() { this.copy = true; }
    void setUnpack() { this.unpack = true; }
    void setInstall() { this.install = true; }
    void setDefragment() { this.defragment = true; }
}
```

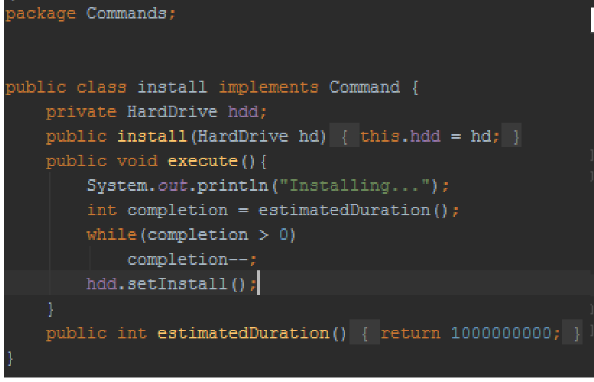
```java
package Commands;

public class Format implements Command {
    private HardDrive hdd;
    public Format(HardDrive hd) { this.hdd = hd; }
    public void execute(){
        System.out.println("Formatting...");
        int completion = estimatedDuration();
        while(completion > 0)
            completion--;
        hdd.setFormat();
    }
    public int estimatedDuration() { return 1000000000; }
}
```



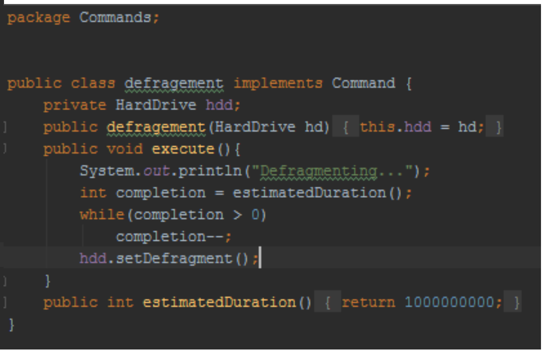
```java
package Commands;


public class copy implements Command {
    private HardDrive hdd;
    public copy(HardDrive hd) { this.hdd = hd; }
    public void execute(){
        System.out.println("Copying...");
        int completion = estimatedDuration();
        while(completion > 0)
            completion--;
        hdd.setCopy();
    }
    public int estimatedDuration() { return 1000000000; }
}
```



```java
package Commands;


public class unpack implements Command {
    private HardDrive hdd;
    public unpack(HardDrive hd) { this.hdd = hd; }
    public void execute(){
        System.out.println("Unpacking...");
        int completion = estimatedDuration();
        while(completion > 0)
            completion--;
        hdd.setUnpack();
    }
    public int estimatedDuration() { return 1000000000; }
}
```



```java
package Commands;


public class install implements Command {
    private HardDrive hdd;
    public install(HardDrive hd) { this.hdd = hd; }
    public void execute(){
        System.out.println("Installing...");
        int completion = estimatedDuration();
        while(completion > 0)
            completion--;
        hdd.setInstall();
    }
    public int estimatedDuration() { return 1000000000; }
}
```



```java
package Commands;


public class defragement implements Command {
    private HardDrive hdd;
    public defragement(HardDrive hd) { this.hdd = hd; }
    public void execute(){
        System.out.println("Defragmenting...");
        int completion = estimatedDuration();
        while(completion > 0)
            completion--;
        hdd.setDefragment();
    }
    public int estimatedDuration() { return 1000000000; }
}
```

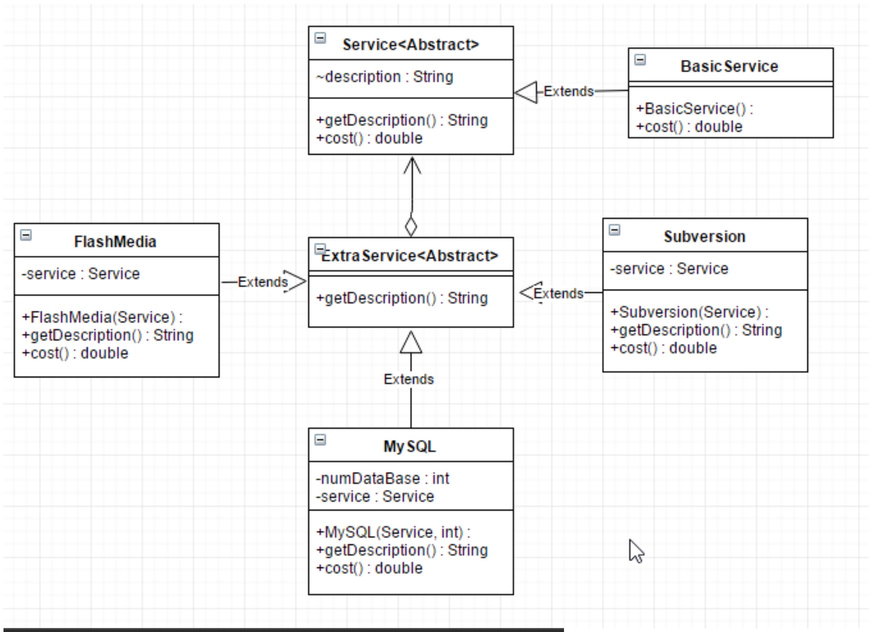
9. (**20 points**) You work for a webhosting company that offers a ton of **hosting** services (Flash Media, MySQL, Subversion) to its customers in addition to a base web hosting plan. You need to write an application that can easily compute the total monthly service fees for each plan. Your application must be able to easily support adding new types of hosting services (such as Ruby on Rails) when they become available.

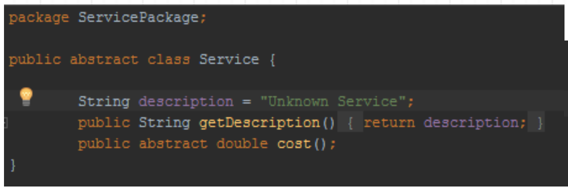
Sample output could be:

```
> Basic Hosting, Subversion Hosting, Flash Media Server Hosting,
MySQL Hosting(w/3 databases): 59.94 a month.
```

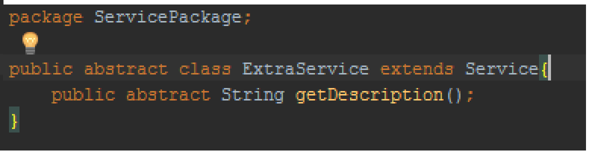
Source : Christmas tree assignment

The pattern most appropriate to solve this problem is the Decorator pattern, based on our decorator assignment involving ChristmasTrees and their prices. This will work the best because you will wrap the price of each hosting service on top of each other to calculate the price of the services as a whole, and are allowed to easily add in new classes to represent new services that won't affect any of the other code.

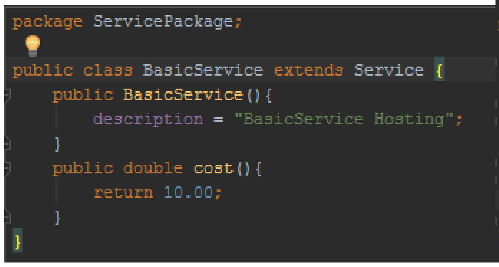




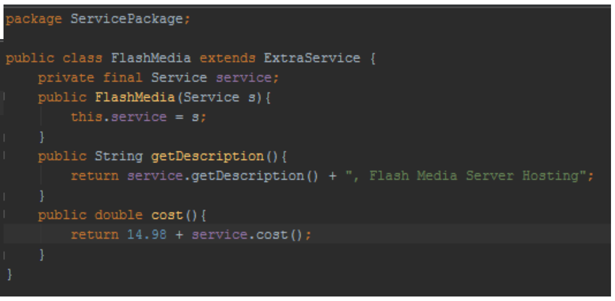
```java
package ServicePackage;

public abstract class Service {

        String description = "Unknown Service";
        public String getDescription() { return description; }
        public abstract double cost();
}
```



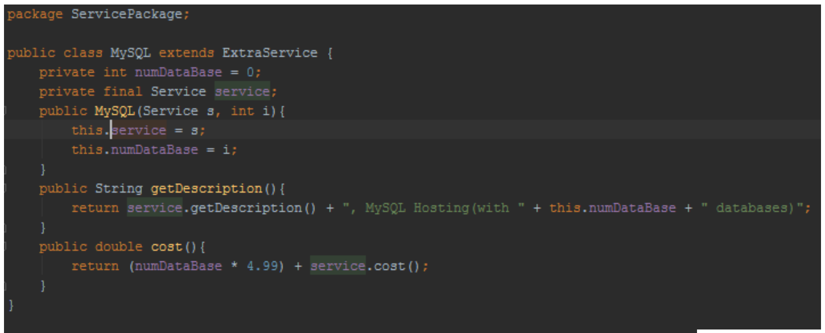
```java
package ServicePackage;

public abstract class ExtraService extends Service{
    public abstract String getDescription();
}
```



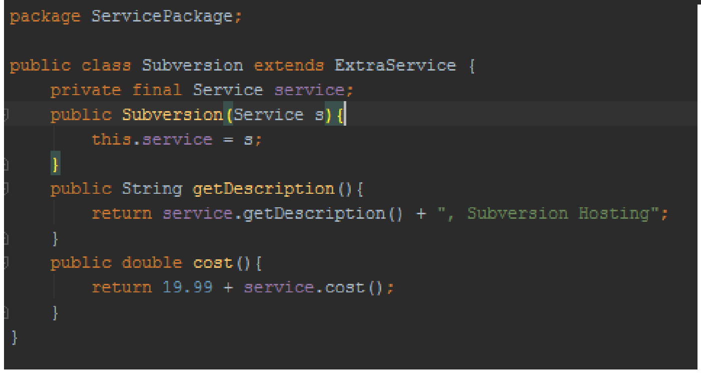
```java
package ServicePackage;

public class BasicService extends Service {
    public BasicService(){
        description = "BasicService Hosting";
    }
    public double cost(){
        return 10.00;
    }
}
```



```java
package ServicePackage;

public class FlashMedia extends ExtraService {
    private final Service service;
    public FlashMedia(Service s){
        this.service = s;
    }
    public String getDescription(){
        return service.getDescription() + ", Flash Media Server Hosting";
    }
    public double cost(){
        return 14.98 + service.cost();
    }
}
```



```java
package ServicePackage;

public class MySQL extends ExtraService {
    private int numDataBase = 0;
    private final Service service;
    public MySQL(Service s, int i){
        this.service = s;
        this.numDataBase = i;
    }
    public String getDescription(){
        return service.getDescription() + ", MySQL Hosting(with " + this.numDataBase + " databases)";
    }
    public double cost(){
        return (numDataBase * 4.99) + service.cost();
    }
}
```



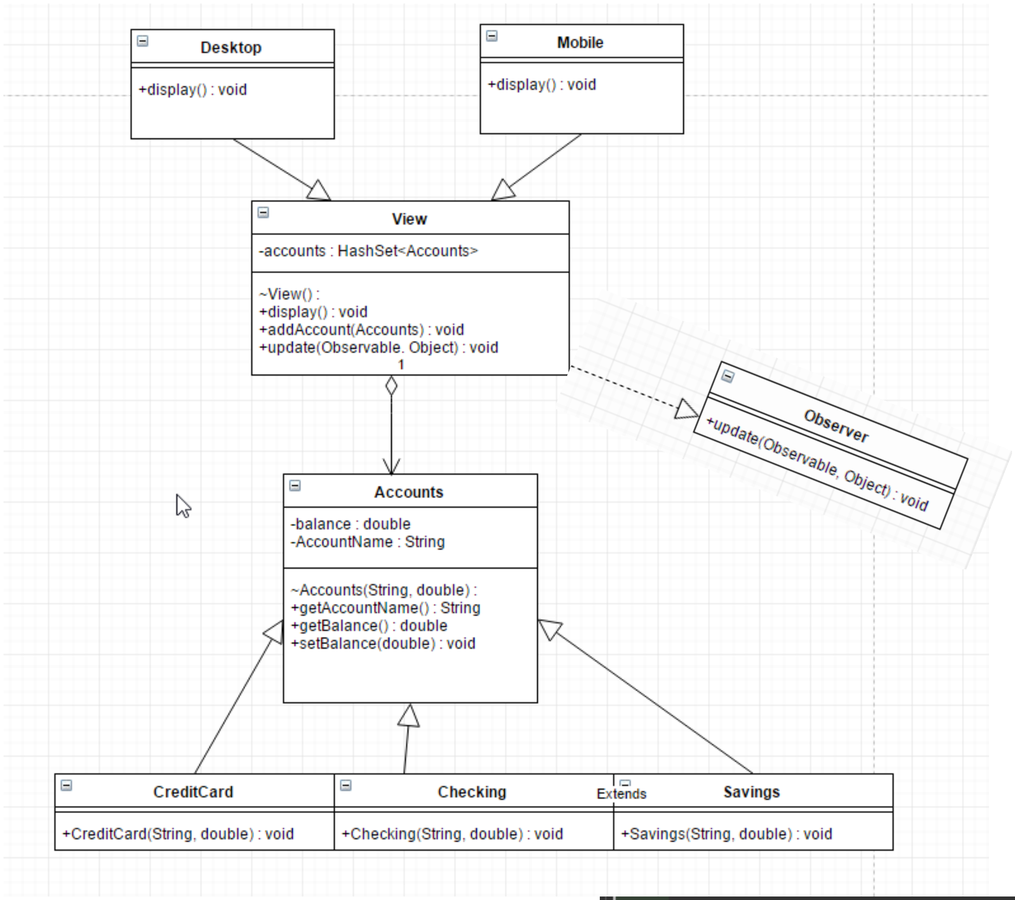
```java
package ServicePackage;

public class Subversion extends ExtraService {
    private final Service service;
    public Subversion(Service s) {
        this.service = s;
    }
    public String getDescription(){
        return service.getDescription() + ", Subversion Hosting";
    }
    public double cost(){
        return 19.99 + service.cost();
    }
}
```

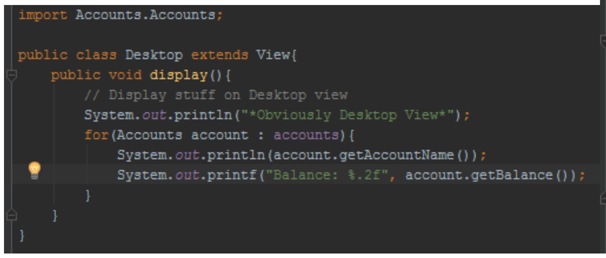
10. (**20 points**). For an online bookstore you need to design a database **engine**. This engine processes transactions that it receives from it's users through a web based interface. A database transaction could be the insertion of a new book, the recording of a sale, or searching for particular books on different fields (title/author). Your engine must keep a list of transactions that need to be executed and cues these as it may take a while for a transaction to complete and transactions are performed sequentially. Should one of these transactions fail, all others can be **reverted** or discarded/undone (called a rollback). For example, if two database tables which refer to each other must be updated, and the second update fails, the transaction can be rolled back, so that the first table does not now contain an invalid reference..

11. **(20 points)**. You have been hired to work for a web-based personal financial management service (like mint.com). Users can get an overview of their financial situation and they can add checking, savings and credit card accounts from different banks that are conveniently compiled into one or more views. Users can access this financial management service either through a desktop application or a mobile device like an iPhone which have different screen sizes and interaction capabilities and hence different views may be required. Bank accounts are checked periodically and whenever a new transaction is detected views must be updated.

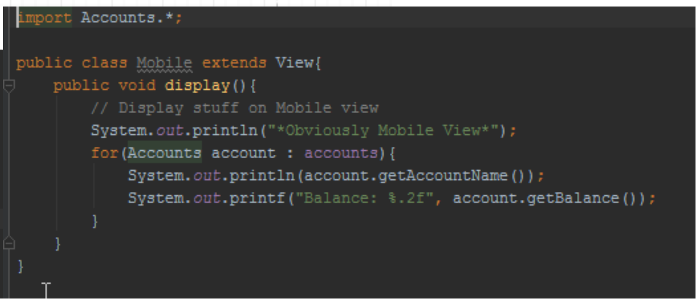
Source : Observer Assignment and http://penguin.ewu.edu/cscd454/notes/Observer.pdf

The pattern most appropriate to solve this problem is the Observer pattern. The observer pattern would let the desktop and mobile versions of the website display up to date notifications on whether an account has been updated, as they are observers of the Observable Accounts.

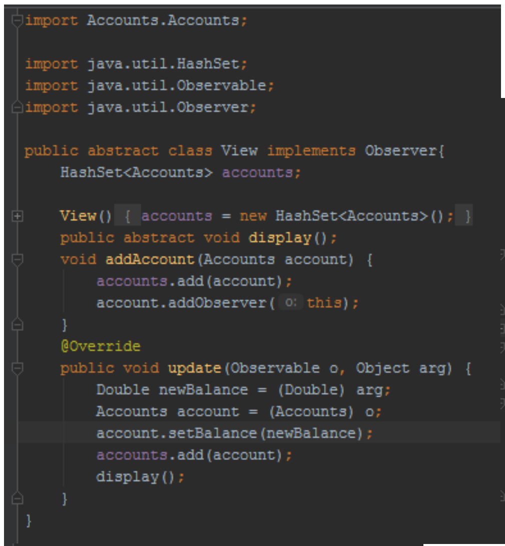

Desktop
+display() : void
Mobile
+display() : void
View
-accounts : HashSet<Accounts>
~View() :
+display() : void
+addAccount(Accounts) : void
+update(Observable, Object) : void
1
Observer
+update(Observable, Object) : void
Accounts
-balance : double
-AccountName : String
~Accounts(String, double) :
+getAccountName() : String
+getBalance() : double
+setBalance(double) : void
CreditCard
+CreditCard(String, double) : void
Checking
+Checking(String, double) : void
Extends
Savings
+Savings(String, double) : void



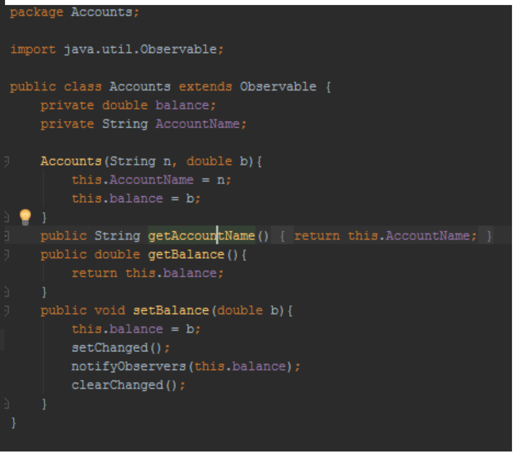
```
import Accounts.Accounts;

public class Desktop extends View{
    public void display(){
        // Display stuff on Desktop view
        System.out.println("*Obviously Desktop View*");
        for(Accounts account : accounts){
            System.out.println(account.getAccountName());
            System.out.printf("Balance: %.2f", account.getBalance());
        }
    }
}
```



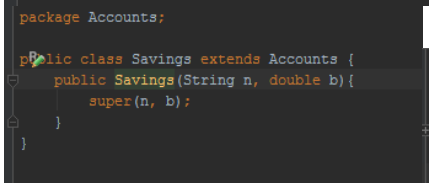
```
import Accounts.*;

public class Mobile extends View{
    public void display(){
        // Display stuff on Mobile view
        System.out.println("*Obviously Mobile View*");
        for(Accounts account : accounts){
            System.out.println(account.getAccountName());
            System.out.printf("Balance: %.2f", account.getBalance());
        }
    }
}
```



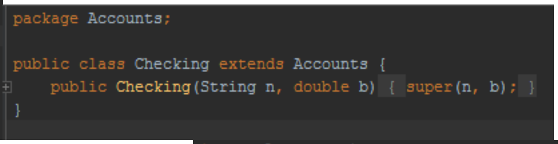
```java
import Accounts.Accounts;

import java.util.HashSet;
import java.util.Observable;
import java.util.Observer;

public abstract class View implements Observer{
    HashSet<Accounts> accounts;

    View() { accounts = new HashSet<Accounts>(); }
    public abstract void display();
    void addAccount(Accounts account) {
        accounts.add(account);
        account.addObserver( o: this);
    }
    @Override
    public void update(Observable o, Object arg) {
        Double newBalance = (Double) arg;
        Accounts account = (Accounts) o;
        account.setBalance(newBalance);
        accounts.add(account);
        display();
    }
}
```



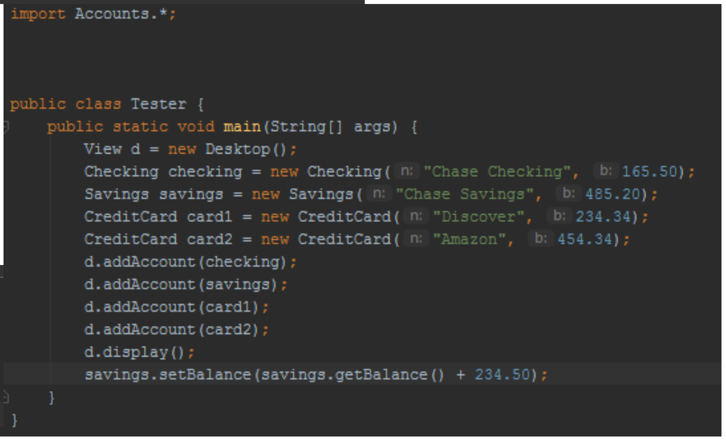
```java
package Accounts;

import java.util.Observable;

public class Accounts extends Observable {
    private double balance;
    private String AccountName;

    Accounts(String n, double b){
        this.AccountName = n;
        this.balance = b;
    }
    public String getAccountName() { return this.AccountName; }
    public double getBalance(){
        return this.balance;
    }
    public void setBalance(double b){
        this.balance = b;
        setChanged();
        notifyObservers(this.balance);
        clearChanged();
    }
}
```



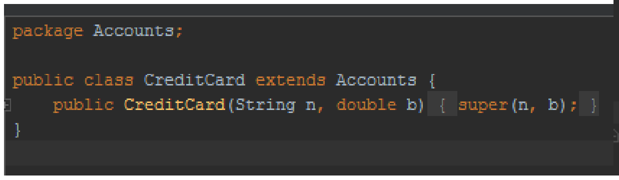
```java
package Accounts;

public class Savings extends Accounts {
    public Savings(String n, double b){
        super(n, b);
    }
}
```



```java
package Accounts;

public class Checking extends Accounts {
    public Checking(String n, double b) { super(n, b); }
}
```



```java
import Accounts.*;


public class Tester {
    public static void main(String[] args) {
        View d = new Desktop();
        Checking checking = new Checking( n: "Chase Checking",  b: 165.50);
        Savings savings = new Savings( n: "Chase Savings",  b: 485.20);
        CreditCard card1 = new CreditCard( n: "Discover",  b: 234.34);
        CreditCard card2 = new CreditCard( n: "Amazon",  b: 454.34);
        d.addAccount(checking);
        d.addAccount(savings);
        d.addAccount(card1);
        d.addAccount(card2);
        d.display();
        savings.setBalance(savings.getBalance() + 234.50);
    }
}
```



```java
package Accounts;

public class CreditCard extends Accounts {
    public CreditCard(String n, double b) { super(n, b); }
}
```

12.(a – **2 points**) What is a non-functional requirement? (b – **8 points**) Choose two patterns and list at least two non-functional requirements that EACH pattern helps with. Justify your choices as necessary.

Sources : http://penguin.ewu.edu/cscd454/notes/IntroToDesignPatterns.pdf
http://penguin.ewu.edu/cscd454/notes/Brendan_Cassida_Design_Patterns_in_Real_Life.pdf

a) A non-functional requirement is a non-behavioral quality of the system that is necessary to produce a proper product.
b) Two patterns and two non-functional requirements that each pattern helps with
    a. Factory Method (Structural)
        i. Security (All data about the different shapes (from our assignment) is separated from the user to protect the code and functionality of the program)
        ii. Interoperability (The different classes that use shape (from our assignment) don't care what shape they get in return, as long as it is a shape and can perform shape calculations)
    b. Command (Behavioral)
        i. Interoperability (if the program I made above were to actually do anything when its installing, it wouldn't care what it was installing or what program was using it. Pass in a file name to install and it will do so)
        ii. Variability (you can put whatever commands you want into the program, as long as they have an execute function)

13. (**5 points**) Describe the difference between published and public with regards to interfaces as discussed by Eric Gamma on the link provided on our website. The discussion references Martin Fowler, who formally identified the difference between the two. NOTE: In describing the difference between two items, you must clearly define what each means, then you can clearly and correctly point out the difference(s).
Source : http://www.artima.com/lejava/articles/designprinciples2.html

Published Interfaces have their own API that can be called upon at any time outside where it is defined without fear of breaking the program.
Public Interfaces are not published interfaces, and when called upon are using internal calls that can break the program if something is called that is outside of the scope of the program.

The difference between the two is, public is only safely accessible within a programs view (within a package or such) and published is accessible from anywhere through the API