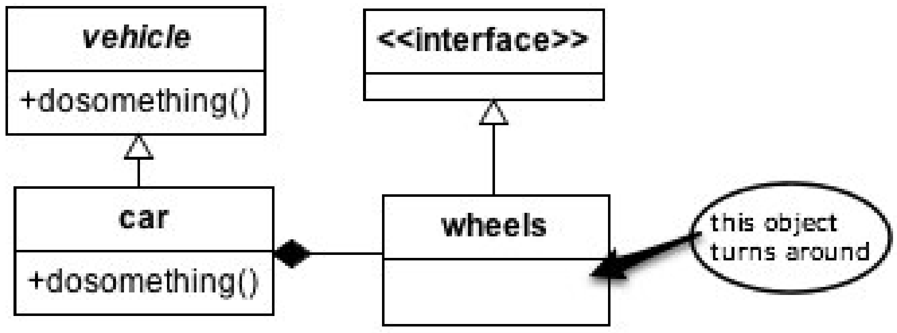
Name: Brandon Fowler____
Midterm
100 points

Rename the file (midterm_Summer14) to include your name (e.g. capaultmidterm_Summer14). Open book, open notes (including our website and the links on it – no Google/Bing searches). Discuss your answers with no one until after the due date. You are on the honor system with regards to these items. You signed a code of ethics document as part of your acceptance into this department. Honor it!

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. The more detail you include, the better chance you have at earning full credit.
NOTE: It is NOT permissible to use a midterm exam for notes from someone you know who took this course previously.



1. **(6 points)** List (at least) three of the most important benefits that design patterns provide to communities of developers?

First, design patterns speed up the development process by providing the re-use of time tested, efficient solutions, to generalized recurring problems. Second, design patterns provide structures that are recognizable between different developers, thus making it easier to interpret and work with another developers code. Third, design patterns provide flexibility to make well structured programs easy to extend, maintain, and update without changing large amounts of pre-existing code in scattered locations.

Source: Penguin site link: http://en.wikipedia.org/wiki/Design_pattern_%28computer_science%29 and my brain.

2. (**4 points**) How are structural patterns different from behavioral patterns, give an example for each type and discuss what system quality these patterns seek to improve.

Structural patterns, focus more on an approach to define relationships between classes in a simple and efficient design, that allows for different pieces to work together. Whereas, behavioral patterns focus on seamlessly changing how objects or groups of objects respond as they are used. Facade is an example of a structural pattern. It improves system quality by providing a single interface, used by the client to manage a complex sub system. This allows the client to do less work; and since the client doesn't have to deal directly with the sub system, it does not need to be updated if the subsystem changes. Template Method is a behavioral pattern, that improves system quality by providing a single interface for many objects, that all implement a set of instructions in the same order. This way the same method calls are made, but behavior can change completely depending on what underlying object is being used. This allows the client to be oblivious to the underlying object and avoid repeated code.

Source: Written class notes, and my brain

---

3. **(5 points)** Explain the reasoning behind the design principle “Tell, don’t ask”. What system qualities does this design principle improve? Illustrate your answer with a simple example (in words/code/UML).

"Tell, don't ask" can improve system qualities in a couple ways. First, it avoids coupling because there is less need to provide access to encapsulated data, for other objects to ask for state. Secondly, it can avoid large and complex switch statements, that check for state and do something based on retrieved information. For example, in our guitar hero assignment we could have used statements like:

```
If(GameCharacter.guitar.compareTo("Telecaster") ==0){
        //play telecaster code
}
else if(GameCharacter.guitar.compareTo("SG") ==0){
        //play SG code
}
else if..........................
```

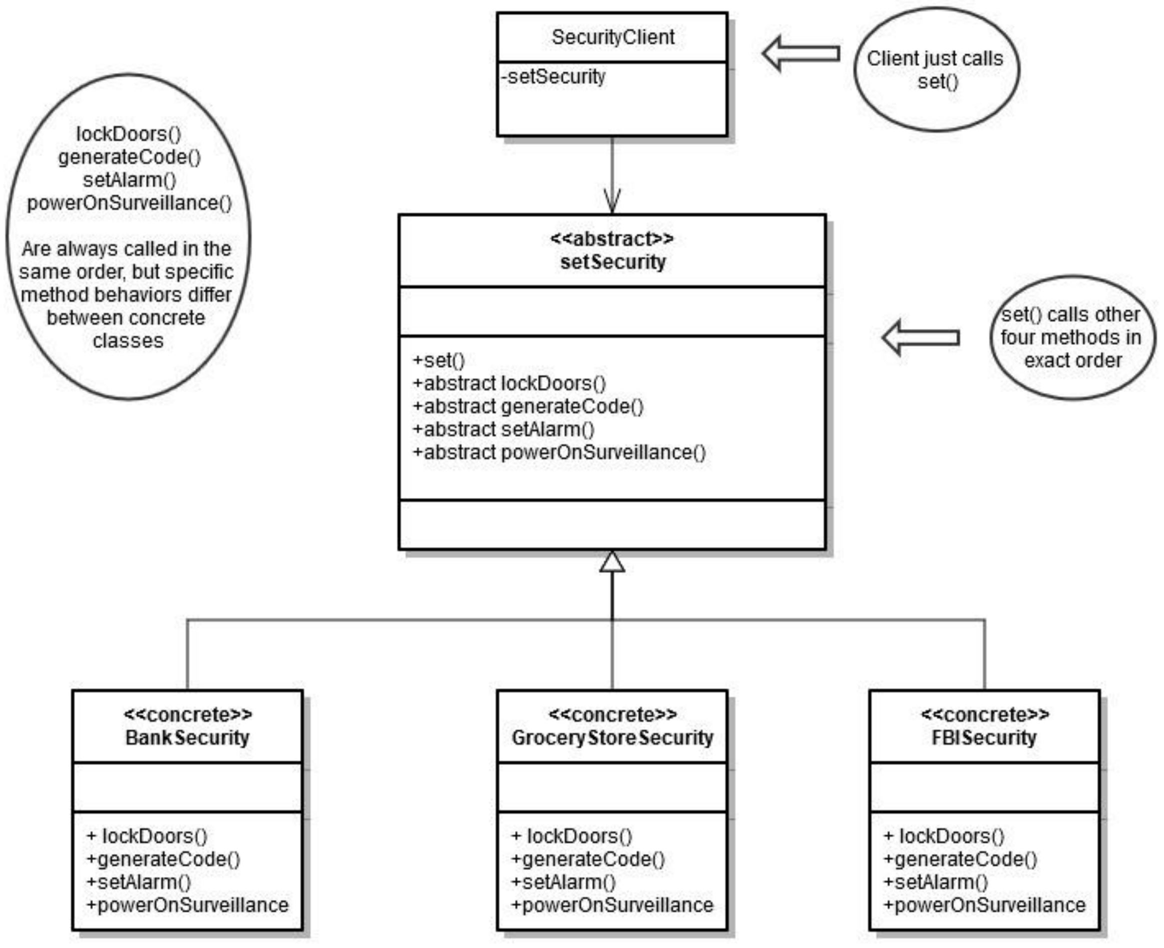
Instead of using this if-else logic that could get long, and need to be updated every time any new behavior is needed, we employed the Strategy pattern, told objects their state, then used an interface to call behaviors, knowing that each object will already know how to behave.

Source: Written class notes, and my brain

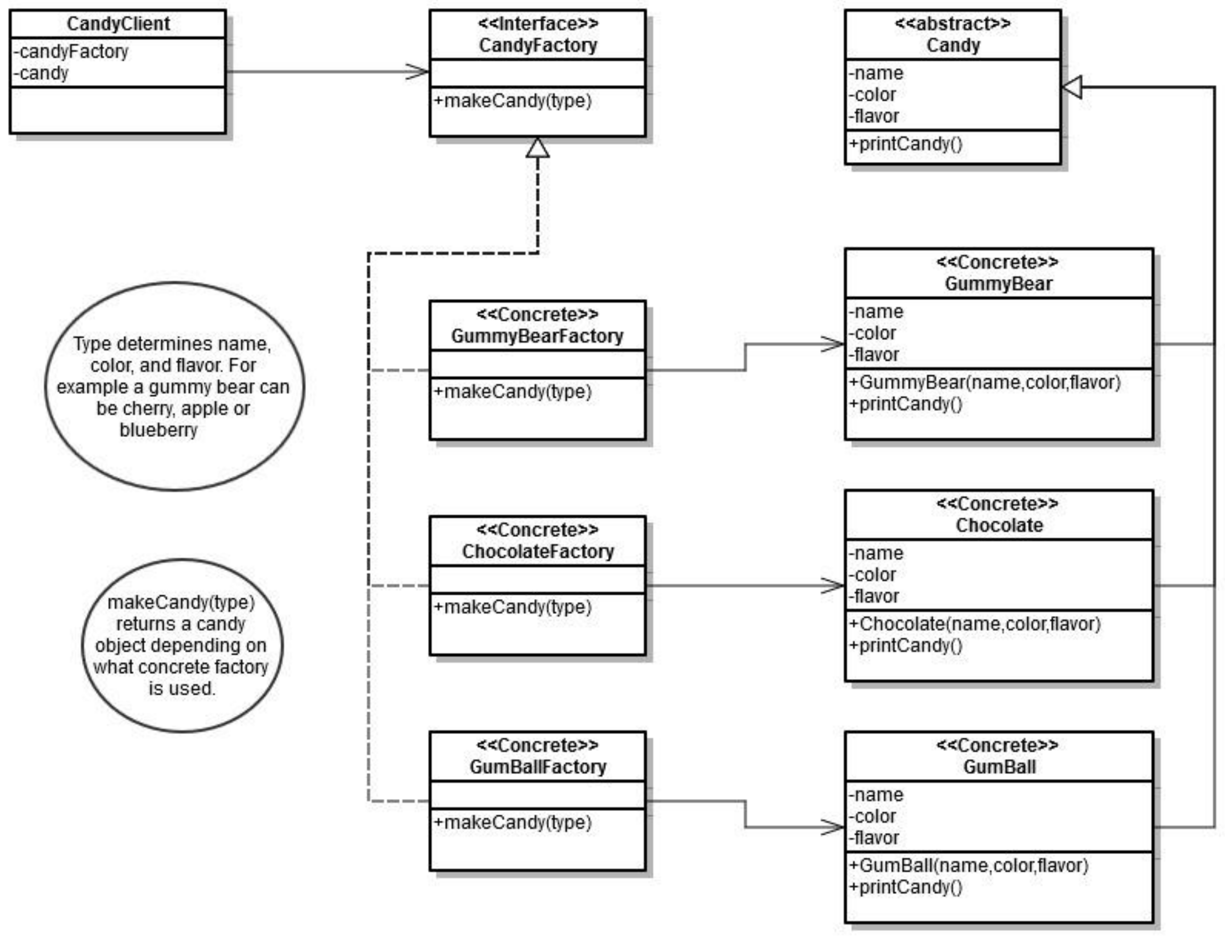
---

4. (**10 points**) Give a class diagram for the **Template Method *and*** the **Factory Method** design patterns. 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. Each pattern should be applied to a specific scenario. The choice of the scenario is up to you!

Template Method Scenario: Security company software:


SecurityClient
-setSecurity
Client just calls set()
lockDoors() generateCode() setAlarm() powerOnSurveillance() Are always called in the same order, but specific method behaviors differ between concrete classes
<<abstract>> setSecurity
+set()
+abstract lockDoors()
+abstract generateCode()
+abstract setAlarm()
+abstract powerOnSurveillance()
set() calls other four methods in exact order
<<concrete>> BankSecurity
+ lockDoors() +generateCode() +setAlarm() +powerOnSurveillance
<<concrete>> GroceryStoreSecurity
+ lockDoors() +generateCode() +setAlarm() +powerOnSurveillance
<<concrete>> FBISecurity
+ lockDoors() +generateCode() +setAlarm() +powerOnSurveillance

Factory Method Scenario: Candy Factory


CandyClient
-candyFactory
-candy
<<Interface>> CandyFactory
+makeCandy(type)
<<abstract>> Candy
-name
-color
-flavor
+printCandy()
Type determines name, color, and flavor. For example a gummy bear can be cherry, apple or blueberry
makeCandy(type) returns a candy object depending on what concrete factory is used.
<<Concrete>> GummyBearFactory
+makeCandy(type)
<<Concrete>> ChocolateFactory
+makeCandy(type)
<<Concrete>> GumBallFactory
+makeCandy(type)
<<Concrete>> GummyBear
-name
-color
-flavor
+GummyBear(name,color,flavor)
+printCandy()
<<Concrete>> Chocolate
-name
-color
-flavor
+Chocolate(name,color,flavor)
+printCandy()
<<Concrete>> GumBall
-name
-color
-flavor
+GumBall(name,color,flavor)
+printCandy()

Example in GummyBearFactory class:

```
Candy makeCandy(type){
        if(type == cherry){
                return new GummyBear(gummy bear, red, cherry);
        }
        else if(type == apple){
                return new GummyBear(gummy bear, green, apple);
        }
        else{
                return new GummyBear(gummy bear, blue, blueberry);
        }
}
```

Source: Penguin site Factory and Template notes, and my brain

5. (**15 points**) (a) List (at least) five OO principles, (b) explicitly describe what each means, and (c) describe a design pattern that employs each principle. Note that inheritance, polymorphism, encapsulation, and abstraction are the Pillars of OOP and not principles in and of themselves. An example of an OO principle is ‘favor composition over inheritance’ – you may not use this as one of your five, BTW.

"Program to an interface, not an implementation", this way clients are unaware of what specific objects they are manipulating, and of the classes that implement the objects. Clients will only know about abstract classes. This allows the client to do less work, and code in the client should not need to change, should the underlying objects be extended or changed. The Facade Pattern implements this principle, by providing the client with a single interface to manage a complex sub system.

"Strive for loosely coupled designs between objects that interact", this way objects know as little as possible about each other, and are not depending on or changing directly accessed data from other objects. This promotes cleaner code, where data only needs to be updated at one point, instead of in multiple objects; and code is less likely to break or develop errors when objects are changed/updated. All design patters should implement this principle. A specific example is Decorator, in which an object is wrapped with new behaviors as needed, without any of the wrapping classes or the wrapped object itself, becoming intimately involved with details of each other.

"Classes should be open for extension, but closed for modification", this means that classes should be developed to be easily extendible via sub hierarchy, to provide new behaviors or function; but the already existing classes should not be modified, as this can cause inconsistency in other classes that are already depending on them; and require allot of messy code updates throughout a program. Factory Method implements this principle by providing an easily extendable interface, through which entirely new objects can be built simply by adding to the structure, with little or preferably no need to update pre-existing code.

"Depend on abstractions, do not depend on concrete classes", this principle walks hand in hand with the previous principle. By depending on abstractions, we are in effect planning ahead by developing easily extensible objects/interfaces, that allow for new features/behaviors without the need for large updates to pre-existing code. Iterator pattern can be an example here, by providing an abstraction with generic behaviors, that other classes can depend on for traversing an "unknown" data structure; and allowing the data structure itself to further define how it is traversed, without revealing its inner workings.

"Only talk to your friends(Law of Demeter)", this means that classes that are in a hierarchy should only make references to other classes that are within one step from them in the hierarchy. A class at the bottom of a hierarchy structure, should not try to "talk" to another class at the top of the hierarchy or vice versa. This promotes cleaner, de-coupled code, that is easier to maintain, understand, and extend. Facade implements this well, as objects that use a large sub-system of classes, only talk to their immediate friend (Facade object), instead of making references to the objects in the sub-system.

Source: Penguin site Quick summery slides, Penguin site specific design pattern links, written class notes, and my brain

---

6. **(15 points) List 5 code smells, describe what each means, and describe how to refactor each.**

   "Duplicate Code", this code smell is in reference to multiple instances of the repeated or very similar code, being used in multiple different classes/objects to perform a similar task. A refactoring solution for this code smell, is to abstract away or encapsulate the code that is the same or performs the same task; so that it can exist and provide functionality from a single area.

   "Conditional Complexity", this code smell refers to large and complex conditional switch statements. This type of code smell can quickly become a difficult to decipher mess, that is arduous to maintain or add functionality. A refactoring approach to this, would be to use "tell don't ask logic", by applying a pattern such as Strategy, Decorator, Command, State or others depending on the goal that needs to be accomplished.

   "Feature Envy", this occurs when a class/object is repetitively relying on a feature/method from another class, to obtain data or functionality. This violates encapsulation, and can cause issues for maintainability and readability. A refactoring solution, could be to move the frequently used feature to the class that is depending on it. If both classes depend on the feature, then it could be moved and encapsulated in into its own class, to provide functionality for both dependant classes.

   "Large Class", this code smell occurs when a class is trying to do too much, and has too many responsibilities. This violates "single responsibility principle", and overcomplicates code. A refactoring approach could be, to simply separate out the responsibilities into reasonable code, that can be encapsulated in separate classes.

   "To Many Comments" , this generally occurs when a developer is not writing intent revealing code, with proper naming conventions, and easily understood logic structures; and thus comments are placed throughout the code to clarify what pieces do. The most obvious refactoring approach to this code smell, is to review code that has been commented; and try to re-write the code itself in a way that is easier understood(naming, structure, intent), so that the comments are no longer necessary.

   Source: Penguin site: http://penguin.ewu.edu/cscd454/notes/SmellsToRefactorings.pdf, written class notes, and my brain.

7. (**5 points**) Assign each design pattern to the right category by placing an 'X' at the right spot. Each pattern belongs to one category. Choose the category that best fits the pattern

Pattern	**Stuctural**	**Behavioral**	**Creational**
Bridge	x		
Facade	x		
Singleton			x
Command		x	
Strategy		x	

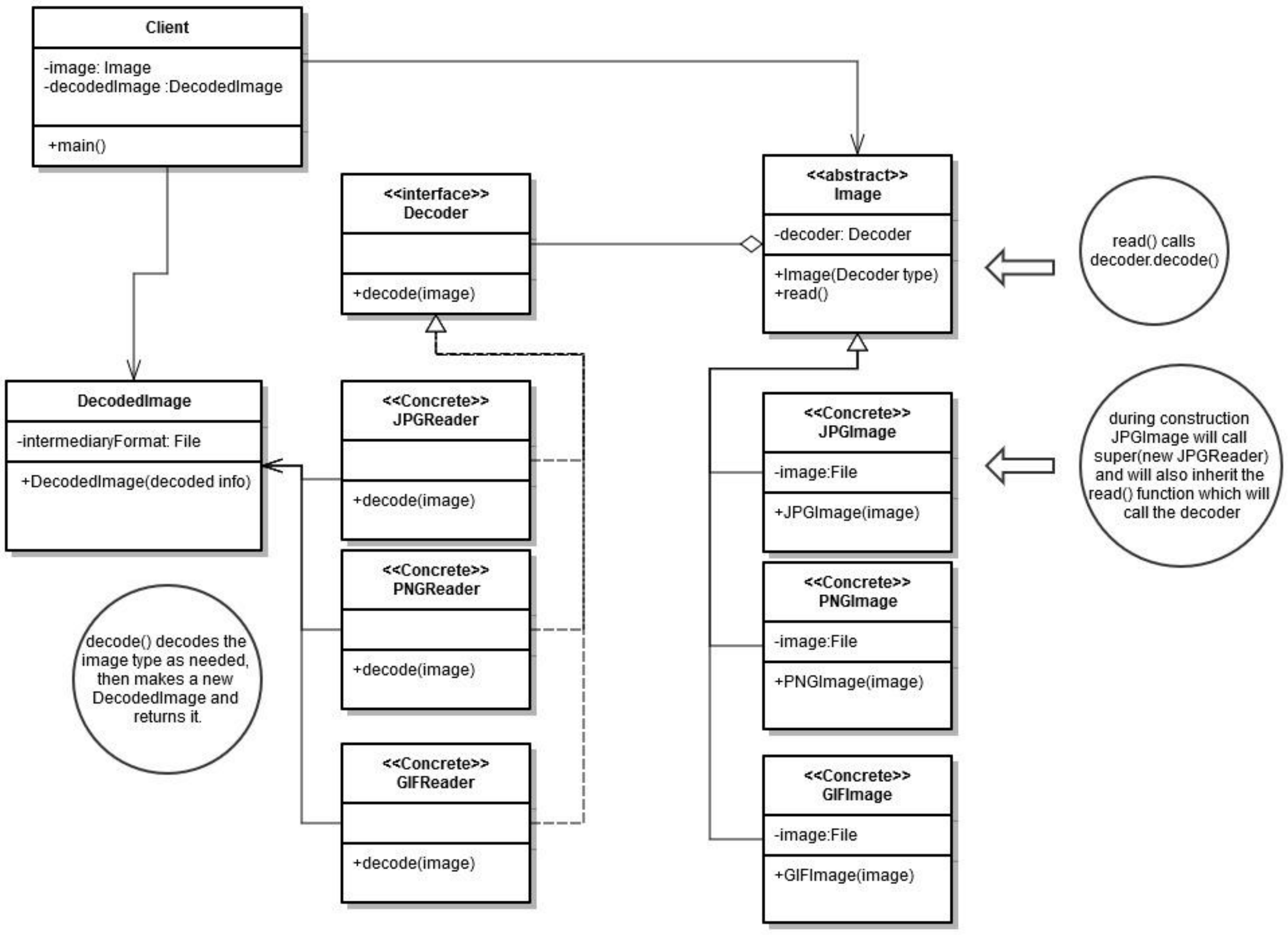
Source: Penguin website diagrams, and course note links for each pattern
Note: The website class notes differ on listing Strategy under structural or behavioral. Behavioral best fits strategy to my understanding.

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 5 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 and enough code fragments to illustrate the implementation of your pattern.

8. **(10 points)**. You are writing a nifty photo application that can read different photos and then creates thumbnails out of them. Your program supports different image formats (JPG, GIF, PNG etc), which are represented by different reader classes for each format that converts the image to an intermediate decoded image format. You anticipate supporting different types of new images in the near future.

I would use the Bridge pattern to solve this problem. Bridge would allow me to have an abstract class for generic images, as well as concrete classes for each type of image format, along with an interface, through which I can provide functionality for decoding each type into an intermediate format. This system would also be easily extendable for new image formats simply by adding new classes.



Simple example:

```
Main(){

        File input = //get new image for decoding(JPG for this example)
        Image image = new JGPImage(input);
        DecodedImage decodedImage = image.read()
}

class JPGImage{

        File image;

        public JPGImage(input){
                super(new JPGReader); //decoder inherited from super class will now decode JPGs
                this.image = input;
        }

        public read(){ //Inherited from Image super class

                return decoder.decode(image);
        }
}
```

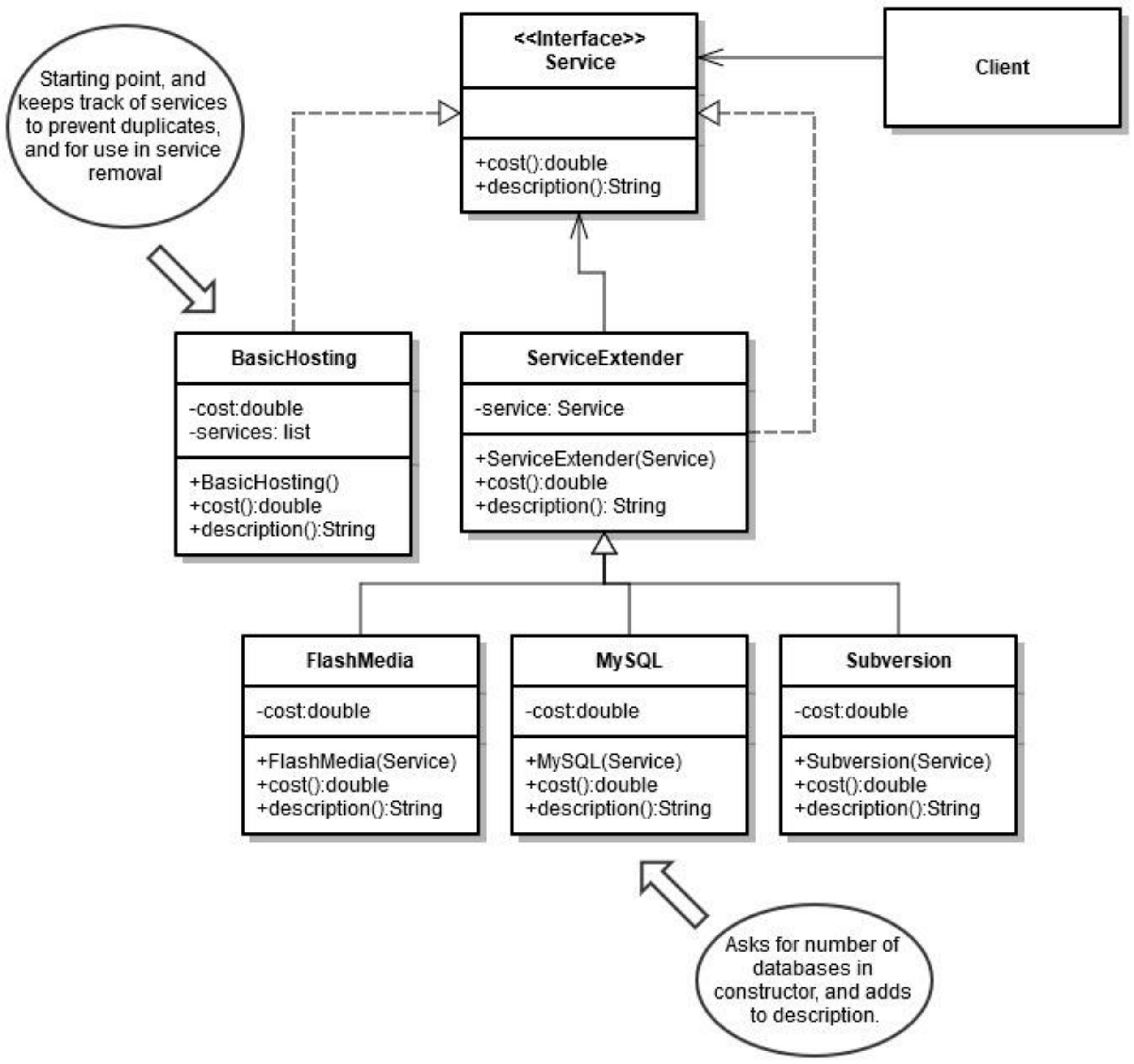
Source: Penguin site Bridge notes, written class notes, and my brain

9. (**10 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.
```

My pick for this problem would be Decorator pattern. Since basic hosting is a requirement, I can take that as my starting point, then through a Decorator interface I can wrap the basic service with further hosting services, and keep track of the total cost. Supporting new services will be as simple as adding one additional class.



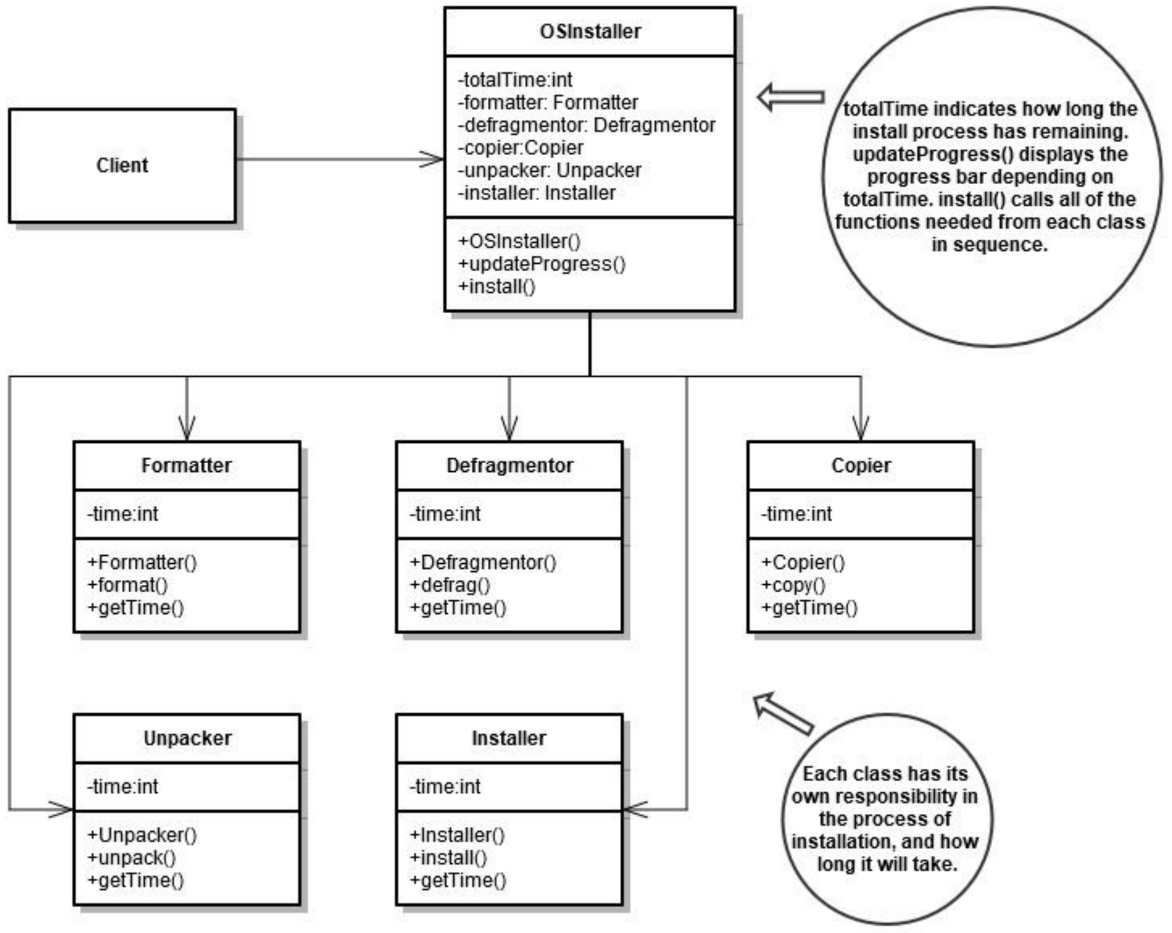
Simple example:

```
main(){

        Service service = new BasicService();
        service = new Subversion(service);
        service = new FlashMedia(service);
        service = new MySQL(service); //Enter 3 for databases

        System.out.println(service.description+":"+service.cost()"+"a month.)
        //This should print the given sample output.
}

//Other classes similar to this one
class FlashMedia{
        double cost = //cost;

        public  FlashMedia(Service service){
                super(service);
        }

        public double cost(){
                return super.cost+this.cost;
        }

        public String description(){
                return super.description()+", Flash Media Server Hosting";
        }
}
```

Source: My written notes, and my brain

10. (**10 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.

Since the installer executes a series of complex tasks in a specific sequence, Facade seems like an appropriate choice. Each complex task can be encapsulated into its own class that the Facade class manages, along with the progress bar.



Example:

```
main(){
        OSInstaller installer = new OSInstaller();
        installer.install();
}
```

```
class OSInstaller(){

        int totalTime;
        Formatter formatter;
        Defragmentor defragmentor;
        Copier copier;
        packer unpacker;
        Installer installer;

        public OSInstaller(){

                formatter = new Formatter();
                defragmentor = new Defragmentor();
                copier = new Copier();
                unpacker = new Unpacker();
                installer = new Installer();
        }

        public updateProgress(){
                //Display bar depending on value in totalTime
        }

        public install(){
                totalTime = formatter.getTime()+defragmentor.getTime()+copier.getTime()
                +unpacker.getTime()+installer.getTime();
                updateProgress();
                formatter.format();
                totalTime = totalTime - formatter.getTime();
                updateProgress();
                defragmentor.defrag();
                totalTime = totalTime - defragmentor.getTime();
                updateProgress();
                copier.copy();
                totalTime = totalTime - copier.getTime();
                updateProgress();
                unpacker.unpack();
                totalTime = totalTime - unpacker.getTime();
                updateProgress();
                installer.install();
                //No need to update time again because install is done
        }
}
```

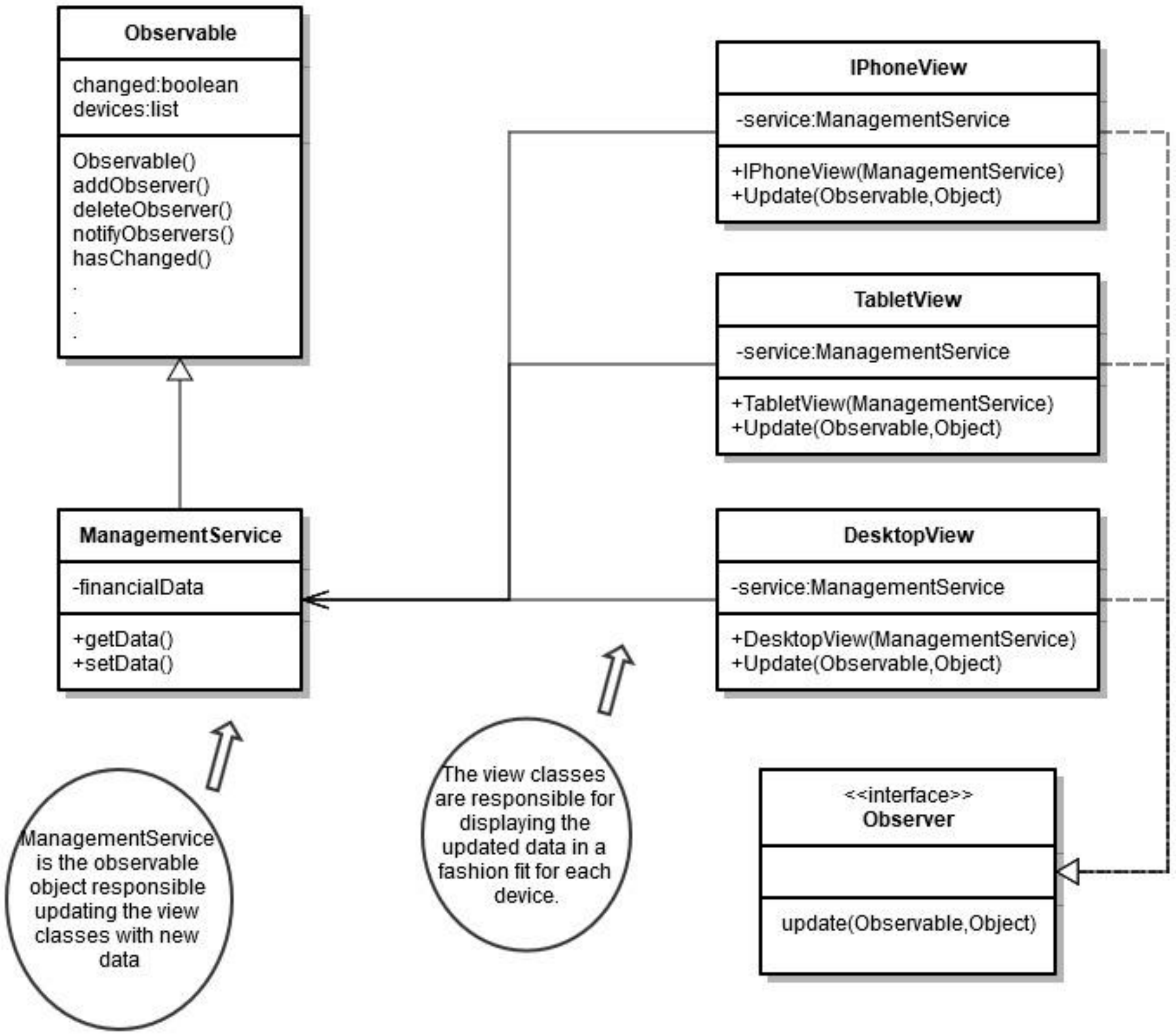
Source: Penguin site Facade notes, and my brain

---

11. **(10 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.

***************Extra Credit Attempt*****************

Since there is a central source of data, and several separate devices that need to be updated based on the data; my solution would be to employ the Observer pattern. The financial management service is the observable object, and the device views are the observers. Every time the management service is updated with new transaction, it can then update the device views.


Observable
changed:boolean
devices:list
Observable()
addObserver()
deleteObserver()
notifyObservers()
hasChanged()
IPhoneView
-service:ManagementService
+IPhoneView(ManagementService)
+Update(Observable,Object)
TabletView
-service:ManagementService
+TabletView(ManagementService)
+Update(Observable,Object)
ManagementService
-financialData
+getData()
+setData()
DesktopView
-service:ManagementService
+DesktopView(ManagementService)
+Update(Observable,Object)
ManagementService is the observable object responsible updating the view classes with new data
The view classes are responsible for displaying the updated data in a fashion fit for each device.
<<interface>>
Observer
update(Observable,Object)

Simple Example:

```
main(){

        ManagementService management = new Management service();
        DesktopView  desktop = new DesktopView(management);

        //Now desktop should be notified and updated any time the management changes
}

class ManagementService(){

        data; //Financial data

        public  ManagementService(){
                //set up initial state
        }

        getData(){
                return data;
        }

        setData(data){
                this.data = data;
                setChanged();
                notifyObservers();
        }
}

class DesktopView(){

        Managementservice service;

        public DesktopView(ManagementService service){
                this.service = service;
                this.service.addObserver(this);
        }

        public update(){
                //Display new data in desktop view
        }
}
```

Source: Penguin site Observer notes, my written notes, and my brain

12. (**6 points**) List three things that cause software to change. With each thing you list, describe an OO principle or pattern that helps with that problem.

    Software can change via patches that are applied in order to fix a bug or issue. A principle that helps with this is, using "Single Responsibility Principle" with proper encapsulation. This way when an issue occurs that needs to be patched, it will most likely be localized in one smaller piece of the software, and can be easily identified and fixed, without major changes to other pieces of the software.

    Software can also change when it is updated with new functionality. This can be made much easier, when a design pattern such as Strategy is employed. Using Strategy, new features/behaviors/functionality can be added to software, simply by adding new classes to the already existing Strategy structure. This means pre-existing code does not need to be changed, and follows the OO principle of "Open for extension, but closed for modification".

    Software can change when developers optimize it for better resource usage. "Strive for loosely coupled designs between objects that interact ", is a principle that can help with this change. If an objects data, functionality or implemented algorithms, need to change in order for more effective resource use; closely coupled objects can form conflictions and failures. If structured in a loosely coupled design instead, then an objects inner workings can change without negative effects to other interacting objects.

    Source: Written class notes, and my brain.

---

13. (**4 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 differences.

    Public means that methods are accessible, and can be called for use, but are subject to change in their implementation. Published means the interface methods are also accessible and can be called for use, but are guaranteed to be dependable. So the difference between the two, is that if a developer is programming to a published interface, he/she is guaranteed stability of methods offered through that interface. Whereas with public, methods are not guaranteed to remain the same, and if code is written to the implementation of the public methods, then dependencies may be broken later as implementations change.

    Source: Penguin site link: http://www.artima.com/lejava/articles/designprinciples.html

---