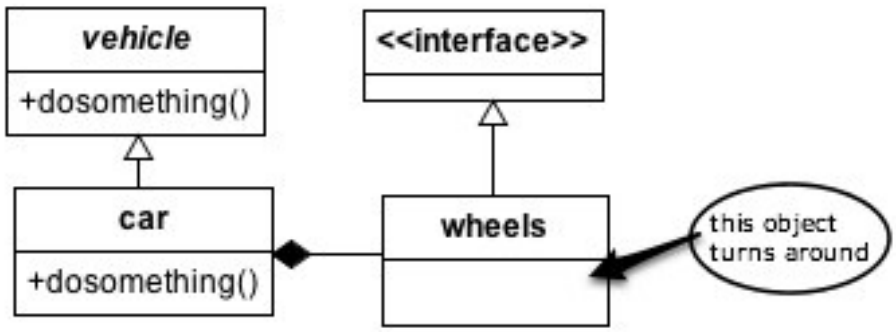
Name_____Julian Welge_____
125 points possible (10 of which are for identifying where answers came from)

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 MULTIPLE 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 below. 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)** List the four pillars of OO. Give a **detailed** description of each pillar.

   The following were notes that I took from the first Panopto recording:

   Abstraction-

   Notation that we have a need to represent some kind of entity in our software solution, but don't want to worry about HOW that entity works on the inside and how it represents data to do the things it needs to do.

   Encapsulation -

class should contain/encapsulate all things necessary to represent and do whatever/ however the class is to store and behave. Data should always be hidden from outside world.

Inheritance -

allows us to build a new, more specialized version of an existing class. Characterized as a " _ is a _ " relationship, ex: circle is a shape. White box - If you're overwriting a method you have to know that you're fulfilling the requirements of a parent method.

Polymorphism -

"many forms", is possible due to inheritance. Allows us to have super/parent classes to any kind of child class. In OO it means that we can treat objects as if they were instances of an abstract class but get the behavior that is required for their specific subclass. <- took this part from the slides on penguin.

2. **(6 points)** What is a design pattern? What benefits does a pattern provide a community of developers?

   From notes I took on the panopto recording: a captured design expertise that solves a certain kind of problem in a software system. From the slides on penguin: Each pattern describes a problem which occurs over and over again in the software development design environment, and then describes the core of the solution to that problem in such a way that the solution can be used again and again, without ever doing it the same way twice.

3. (**6 points**) Why does inheritance violate encapsulation principles?

   Because relevant data can be stored outside the scope of the class inside parent classes therefore no longer keeping that information incapsulated. From welcome to design patters slides on penguin.

4. (**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).

From SOLIDPrinciples.txt on canvas
Single Responsibility - classes should do 1 thing only. Everything inside the class should be only things specific to that class. If the fields and behaviors are common among multiple classes then they belong in a super class or an interface/abstract class of some sort. A singleton design of a static class is an easy example as it's data is only stored in those specific fields.

Open/close - a class should be open for extension but closed for modification. The initial point of writing the code is writing algorithms that can with stand change and if you're constantly having to change your code depending on particular circumstances then it's not very robust code. Solid code would be code that would work for lots of other code to be extended off of (open for extension) but would require no editing to actually be done to it (closed for modification), all of the design patterns are meant to follow this, the factory idea has been the most interesting so far with the pizza example from the book. Or decorator and how everything is the type of the the super abstract class.

Liskov - sub classes should operate the super class has intended them to operate. That way when you call a method of a superclass that is overwritten by subclass you still get the right

information. This allows for accurate information to be accessible when utilizing polymorphism. The decorator design was a cool example of the method cost() having to return it's cost and then automatically appending to the rest of the cost() called before hand at runtime.

Interface Segragation - Instead of having one big interface with a lot of methods, break it all into a bunch of smaller ones while making sure methods that should work together are kept together. Keeps code loosely coupled and highly cohesive. The strategy assignment with Guitar Hero was a good example of this as we split up all of the behaviors.

Dependency inversion principle - To eliminate concrete references to concrete classes, if there are things in common between the classes then they should be moved up into a superclass. Gets rid of duplicate code and also makes it easier for people who want to use the code as they can use a reference to the super class and not have to worry about implementing the sub classes. The factory work we did was an example of making code loosely coupled by putting the stuff in common in a super class and the specifics up to the factory to decide.

5. **(15 points)** List 5 code smells, describe what each means, and describe how to re-factor each.

From smells to refactorings on penguin site:

Duplicate code - can be explicit or subtle, explicit is just obvious copying of code while subtle is code that seems like it's doing different things but is basically doing the same thing. Re-factoring duplicate code can be done by extracting commonalities in the code and moving them up to implement them in a more abstract way.

Free loader - a class that isn't doing enough to pay for itself (lazy class), either eliminate it or use as a single static class.

A large class - a class with too many variables and responsibilities. Too many instance variables could be an indication that the class is trying to do too much and can be split into different classes. Extracting behaviors into interfaces and trying to extract any sub classes is a start to solving this issue. Document also mentions replacing data value with Object, replacing implicit language with interpreter, and replacing state-altering conditionals with state.

Divergent change - This happens when a class is commonly changed in different ways for different reasons. Can fix this by separating the divergent responsibilities to decrease the chance that one change could affect another and lower maintenance costs. (split up the classes).

Data clumps - bunches of data that hang around together throughout the program that could be their own object probably should be. Can fix this by making an object of the clump of data and passing just that object as a parameter.

6. (**5 points**) Describe coupling. Describe cohesion. Which combination of coupling and cohesion is most desirable?

The ideal code is loosely coupled and highly cohesive. If code is loosely coupled, then it is not very dependent on other pieces of code in order to be utilized. By lacking dependency on other code, code would be highly cohesive in that it can easily be plugged in to perform operations in other projects. Highly cohesive code however still needs to do/represent something that is worth actually implementing, just because you can import a chunk of code into a project to do one thing, doesn't mean it's necessarily efficient if it also means you're importing a bunch of unused code along with it.

Design Problems

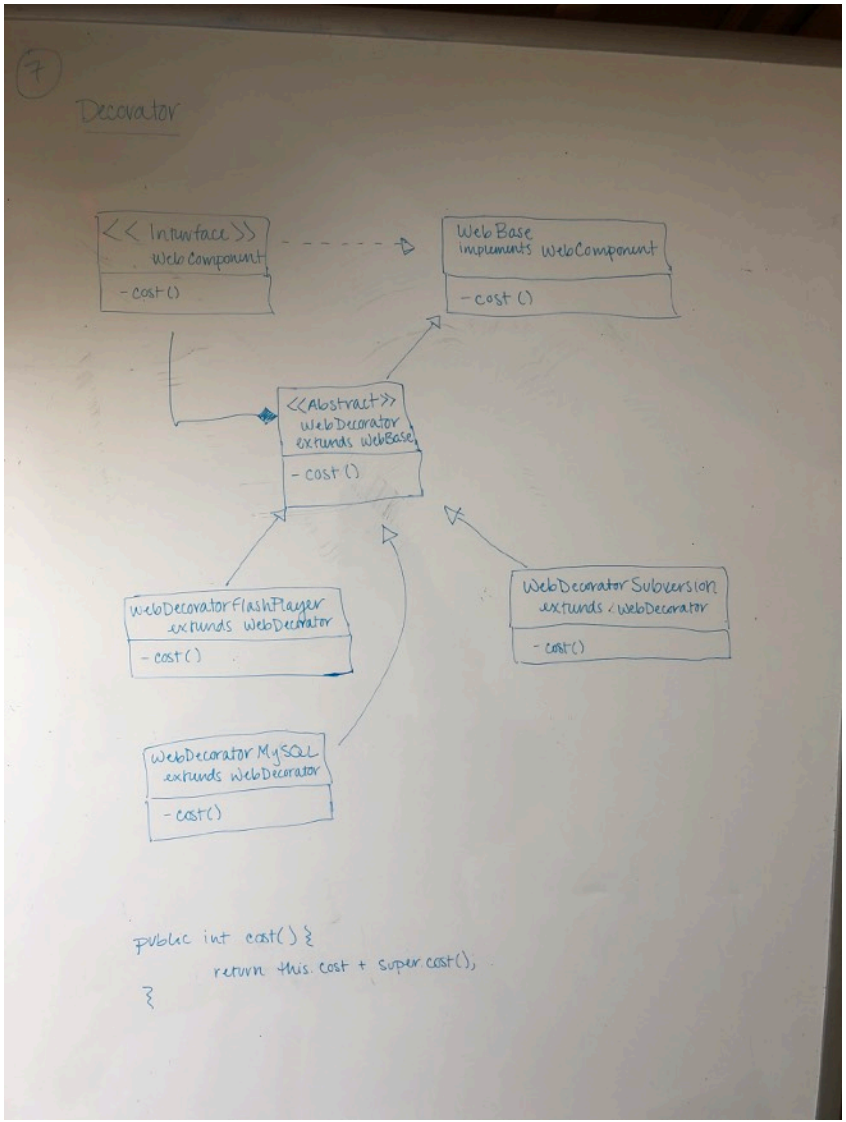
Three design problems are listed on the pages that follow. Select the most appropriate design pattern to use to address the problem and clearly motivate how it addresses the problem. **Furthermore**, show an appropriate class diagram followed by enough code fragments to illustrate the implementation 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.

7. (**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.

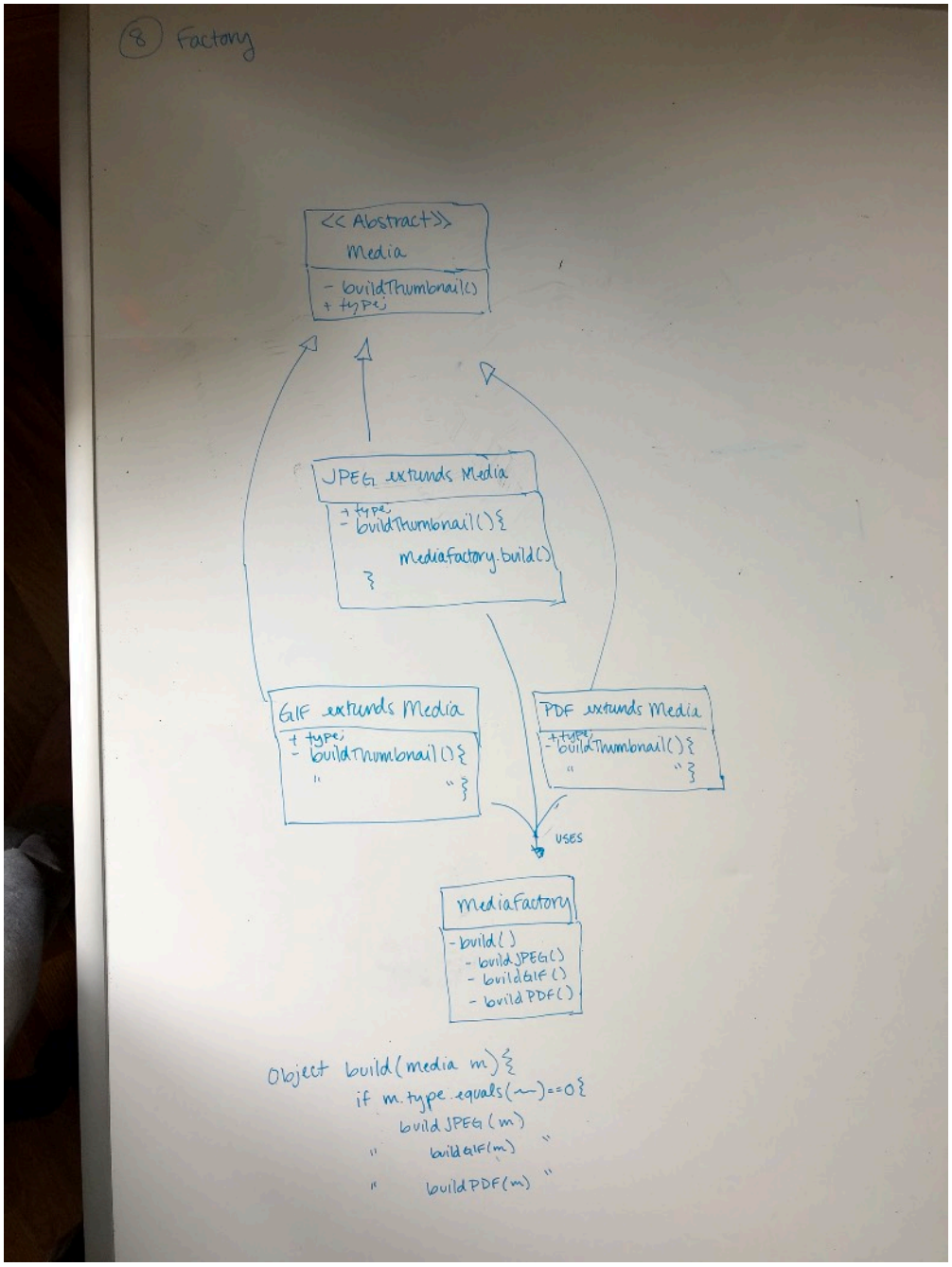
Sample output could be:

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



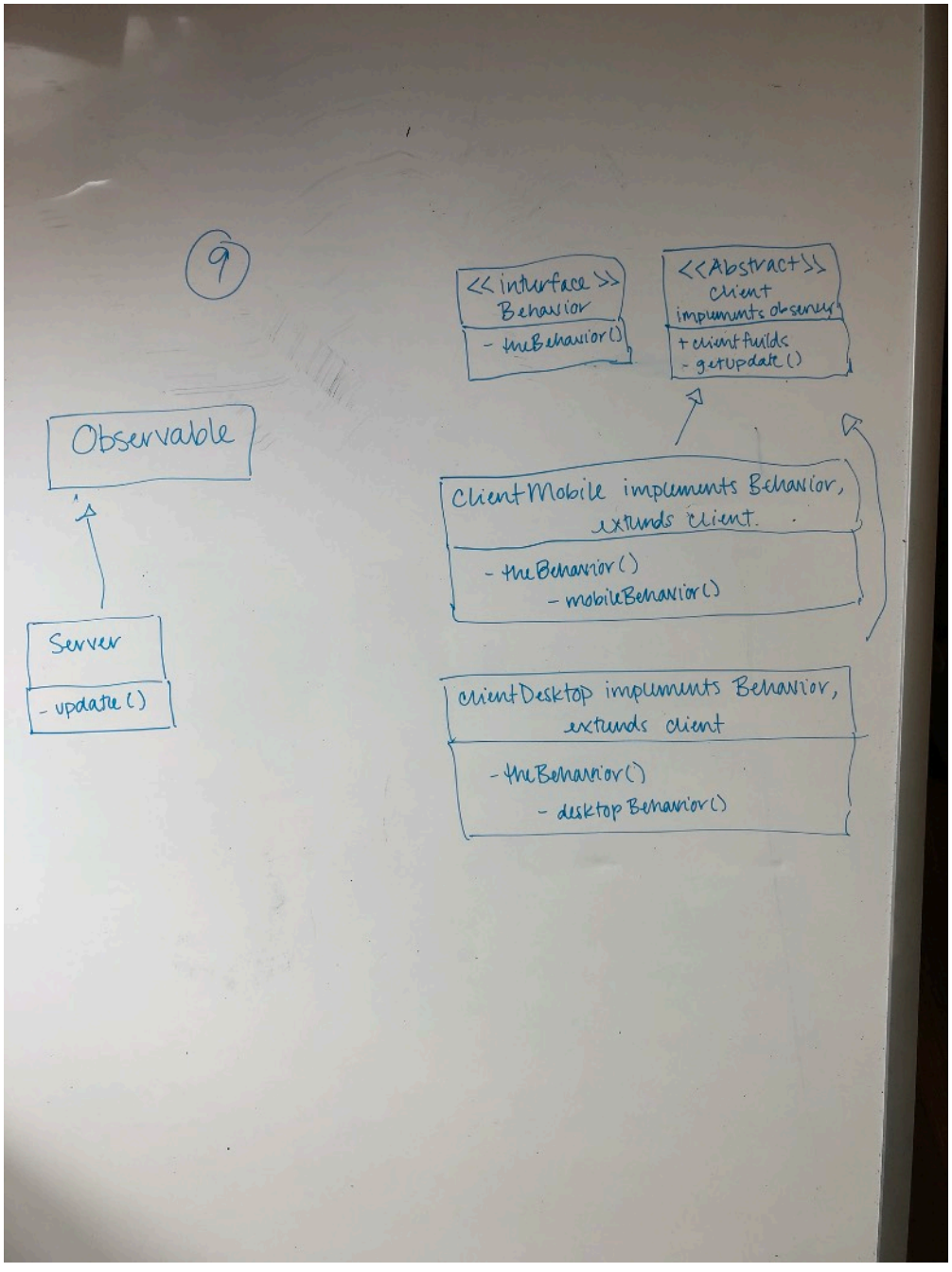
You could use a decorator design. It's helpful when totally costs of products as each class would call a cost() method that would append that classes cost to other cost()'s called at runtime.

8. **(20 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.



With a factory design, you could determine the type of file you're working with at runtime inside a separate factory class. To add more types you would need to add that constructor to the factory but saves time and code when compared to keeping it all encapsulated in one class.

9. **(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.



I think the bridge pattern on the penguin site is similar to the strategy we first did, where we can separate the behaviors with interfaces and an abstract class so that different types can work independently (mobile/desktop), also with notifying people would require using something like the Observable class and having each client implement Observer.

10. (**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).

Public can be thought of what's really available to the public in the code vs. published which would be all that the API says is in the code. Something could be public and available, but that doesn't mean that it's been published. If you show that you're implementing an interface and then call a group of methods beyond the interface in that same class, it can be confusing even to what level these methods exist or if they exist and you won't know until a problem occurs that breaks your code.

EXTRA CREDIT: Read the slides on the penguin class website titled "Brendan Cassida Talk Notes" then answer the following questions. (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.