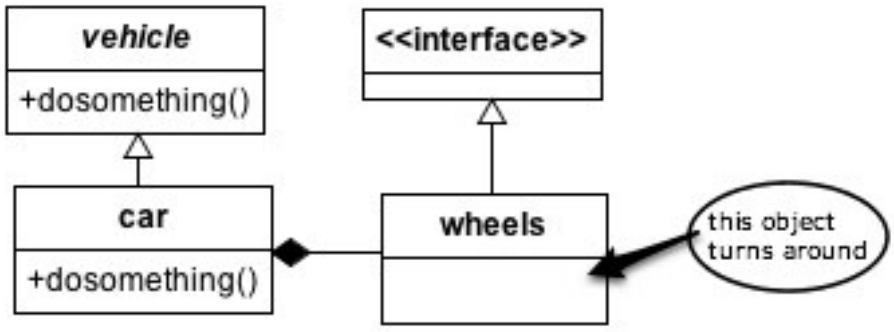
Name_____**Ian Kaiserman**_____
125 points possible (10 of which are for identifying where answers came from)

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.

   Abstraction – a defined entity that we want in a piece of software, but we don't want to worry about how it works until it needs to be implemented (pairs with inheritance)

   Encapsulation – A class should contain all fields and behaviors that are directly relevant to that class type and nothing more. The way these behaviors are implemented are not important to outside classes.

   Inheritance – an "is a" relationship, allows us to create specialized, or "child" versions of a class.

Polymorphism – allows a super class to have many different forms, made possible due to inheritance.

Source: Notes

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

   A design pattern is a widely known and accepted, reusable strategy of structuring code in a software that is used to solve a variety of problems in software related to its functionality and ability to adapt to change.

   Source: self

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

   Inheritance violates encapsulation principles because the child classes need to know about how the parent class works in order to function, which goes against the concept of outside classes not needing to know how methods are implemented

   Source: self

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).

   Single Responsibility: Every class or method should be responsible for performing one singular function or a singular set of directly relevant functions. Example: Strategy pattern

   Open/Closed: Classes should be open for extension but closed for modification. Classes should be able to have functionality added to them without having to directly modify the code. Example: Decorator pattern

   Liskov Substitution: Tests whether a child class of a parent class can be substituted with another child class without breaking functionality. If it breaks functionality, then the child class that causes issues needs to be abstracted differently to make up for that behavior. Example: Strategy pattern

   Interface Segregation: Sets of behaviors should be split up in such a way that the client only knows about things it needs direct access to. Example: Observer pattern (using getters/setters to obtain info)

   Dependency inversion: subclasses should always and only depend on superclasses, not the other way around. Abstract classes and classes at the top of a hierarchy should be completely independent of everything else.

   Source: notes

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

   Comments: comments should explain WHY a functionality exists, not what it

does

Duplicated code: Duplicate code should be isolated and turned into its own method/class to avoid repetition and wasted time/space and increase readability

Dead code: Avoid code that never gets reached in an actual program. Remove it.

Long method: Avoid unnecessarily long methods that perform multiple functions inside of it. Split it into multiple, smaller methods that each do it's own unique, singular task

Long parameter list: Like long method, if a method has numerous parameters, remove any unnecessary ones or combine them into singular objects

Source: notes

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

   Coupling describes how dependent different classes and class structures are on each other. This also influences how much code would need to be changed in different classes if functionality of one class was changed, added to, or removed.

   Cohesion describes how unified a code structure is. Code that is dependent on each other or part of the same "group" of functions (for example, a class) should all be extremely relevant to each other.

   The best combination is loose coupling and high cohesion. Independent objects with good relevant organization of those objects and behaviors.

   Source: notes

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.

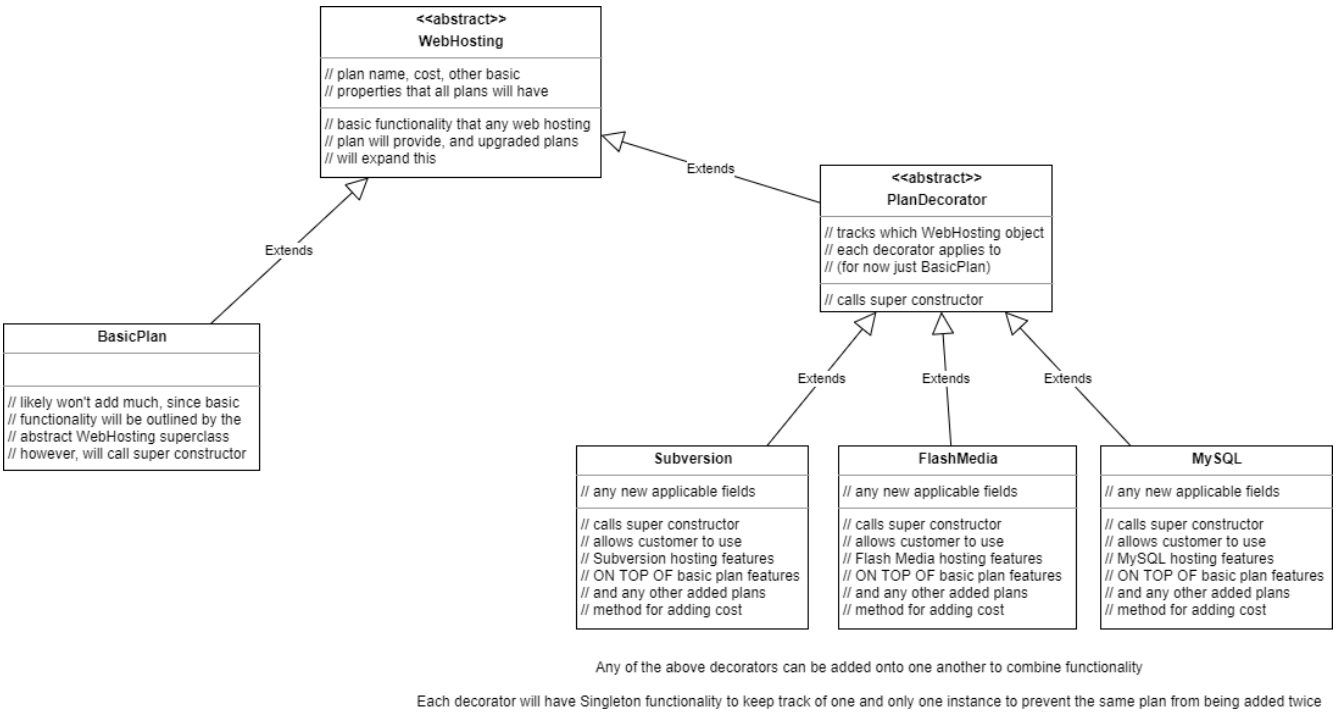
To clarify, you will do **three** things for each problem. **First**, list a pattern that you think best solves the problem along with justification for why you chose it/ why it works. **Second**, draw UML that represents that pattern in the context of the problem. **Third**, include code fragments/snippets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

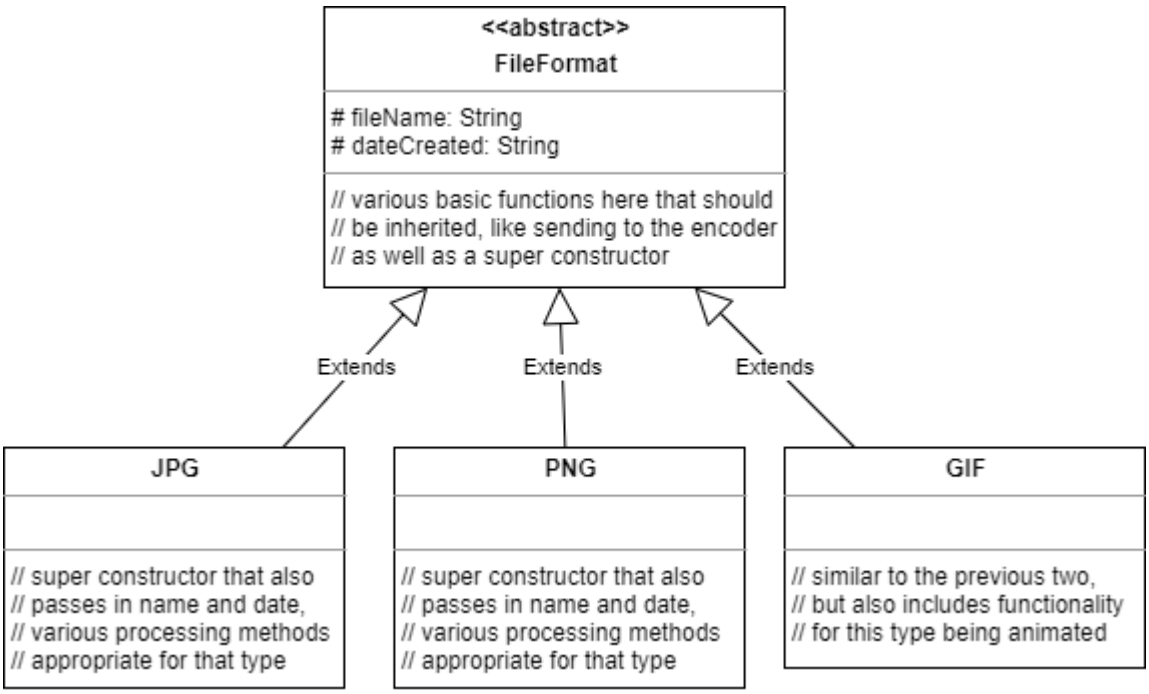
```
> Basic Hosting, Subversion Hosting, Flash Media Server Hosting,
MySQL Hosting(w/3 databases): 59.94 a month.
```

The pattern I would choose for this solution would be Decorator. Essentially, customers can select one of several plans, but all of them build off a standard hosting plan. The most efficient way of creating this solution is making an abstract Decorator class with basic, shared functionality. Then create the decorators themselves which are the "upgraded" hosting options, these would extend the Decorator class mentioned before. Then, there would be an abstract WebHosting super class that the Decorator class, Basic Hosting class, and all concrete decorators would extend. Each concrete decorator would add more functionality that the customer has access to, including adding the price to the current price. The concrete decorators' classes should also be created with Singleton functionality to prevent the same plan from being added more than once, as that would obviously be pointless. This solution not only allows more types of upgraded plans to be added, but also allows a larger variety of basic plans to be created as well to suit all customers' possible basic needs.



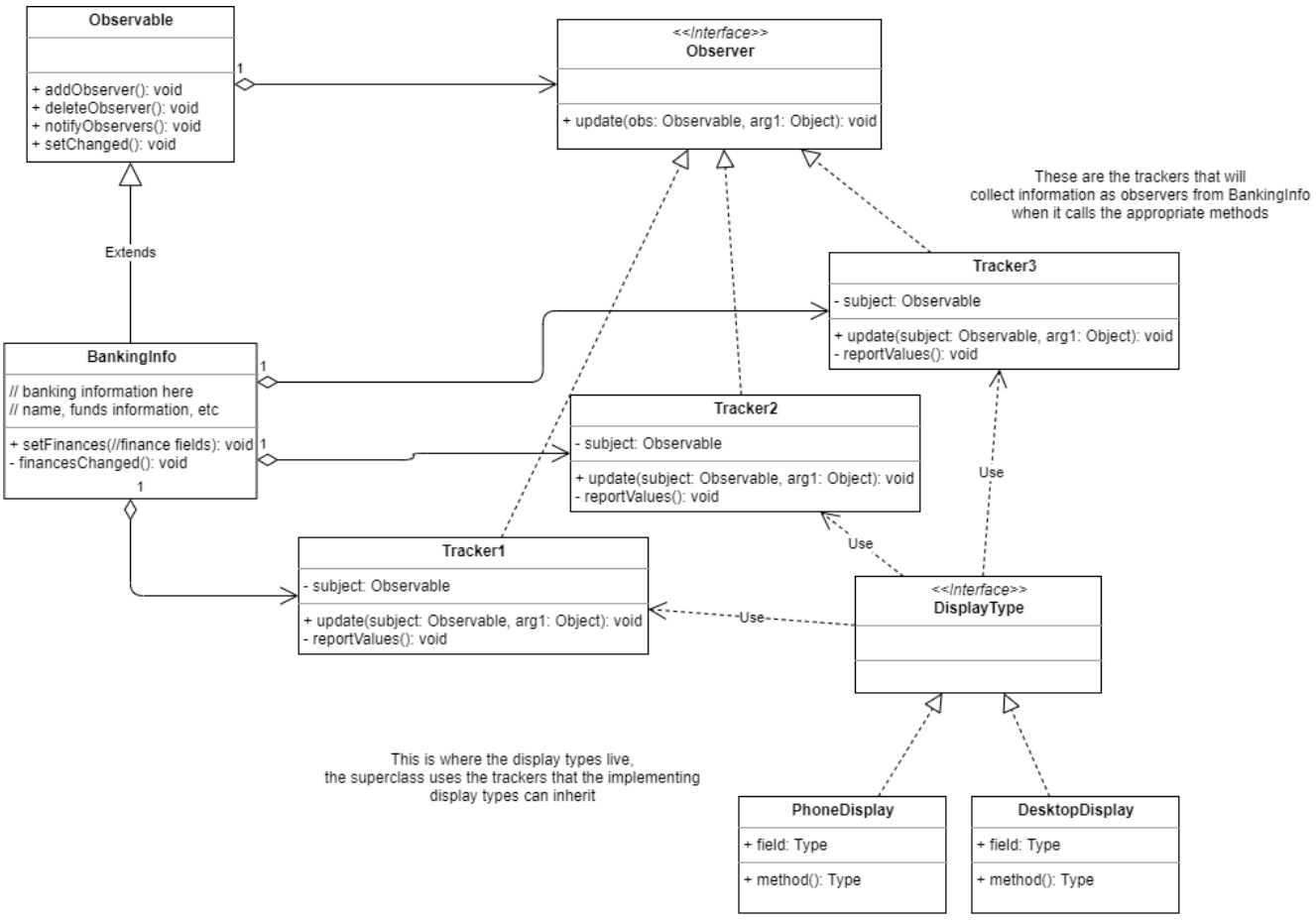
8. **(20 points)** You are writing a nifty photo application that can read different photos and then **creates** thumbnails out of them. Your program supports different image formats (JPG, GIF, PNG etc), which are represented by different reader classes for each format that converts the image to an intermediate decoded image format. You anticipate supporting different types of new images in the near future.

The pattern I would choose for this is Strategy. Like the duck example from the textbook, and the guitar hero assignment, the user can select what type their file is and upload it. Each file type would have its own class that inherits basic fields and methods (like storing a file name and date for example) from a super class FileFormat. When the user selects a file format that they want, it creates a new instance of that class that has methods to be able to process that image type appropriately. This would also allow for expansion, as new file formats would simply be a new class that extends the FileFormat super class and add its own methods for its processing functionality.



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.

The best solution for this problem would be observer. The reason is that elements of this website need to be updating in such a way that whenever the user checks the site, they always see the most up-to-date information on their account. Updating the different fields would be a simple observer pattern, with the subject or observable being the source of the banking information and the different display elements being the observers. However, multiple classes will be needed to process the display elements differently to account for different devices (phones, PCs, etc)


Observable
+ addObserver(): void
+ deleteObserver(): void
+ notifyObservers(): void
+ setChanged(): void
1
<<Interface>>
Observer
+ update(obs: Observable, arg1: Object): void
Extends
These are the trackers that will
collect information as observers from BankingInfo
when it calls the appropriate methods
Tracker3
- subject: Observable
+ update(subject: Observable, arg1: Object): void
- reportValues(): void
BankingInfo
// banking information here
// name, funds information, etc
+ setFinances(//finance fields): void
- financesChanged(): void
1
1
1
Tracker2
- subject: Observable
+ update(subject: Observable, arg1: Object): void
- reportValues(): void
Use
Use
Tracker1
- subject: Observable
+ update(subject: Observable, arg1: Object): void
- reportValues(): void
Use
<<Interface>>
DisplayType
This is where the display types live,
the superclass uses the trackers that the implementing
display types can inherit
PhoneDisplay
+ field: Type
+ method(): Type
DesktopDisplay
+ field: Type
+ method(): Type

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).

    According to Erich Gamma, the main difference between public and published is its intent to be used. Something can be publicly accessible, but that does not mean it is ready to be used yet. When someone marks something as published, that usually means it is in its final form for intents and purposed for the time being. Using something that is public but NOT published leaves the risk of something breaking/being changed before it is published.

    Source: link provided

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.

A non-functional requirement has to do with the quality of a system, rather than actual functionality. It is these non-functional requirements that help determine what functionality should be used in a system before it is implemented.

Two patterns that can be examples of this are Singleton and Decorator. The purpose of singleton is to only keep one instance of a class object in memory and no more. The non-functional requirements that can prompt the use of this are reusability of that instance, and transferability of that instance to different parts of the program. For decorator, the use of this pattern can be prompted by needing extensibility so that more "add-ons" can be created and implemented with ease, as well as basic simplicity, solving the problem of class explosion if a number of these decorators exist.