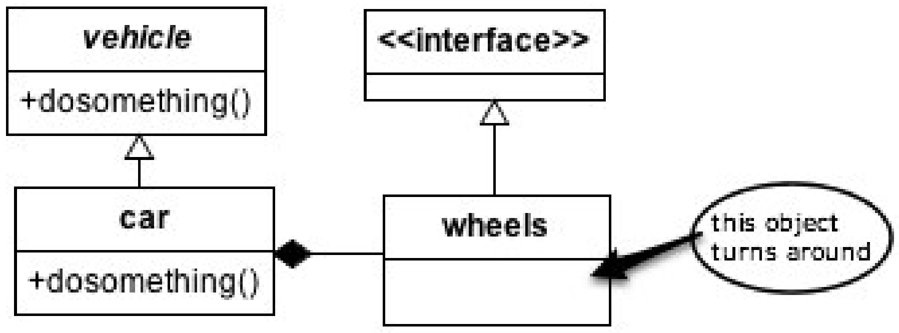
Name: Ethan Tuning

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 7 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)** List (at least) four of the most important benefits that design patterns provide to communities of developers.
.

Design Patterns provide:

1. A tried and true way to solve common problems in OO design.
2. A standardization at which to approach software creation.
3. A means to build a common vocabulary around development.
4. A means to deploy the intent/purpose of OO. In other words, take advantage of the power and potential that OO design provides for us.

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

It tightly couples the parent and child classes. The child class has knowledge about the private information within the parent class. This is not a bad thing in some cases, but most of the time you should favor composition over inheritance.

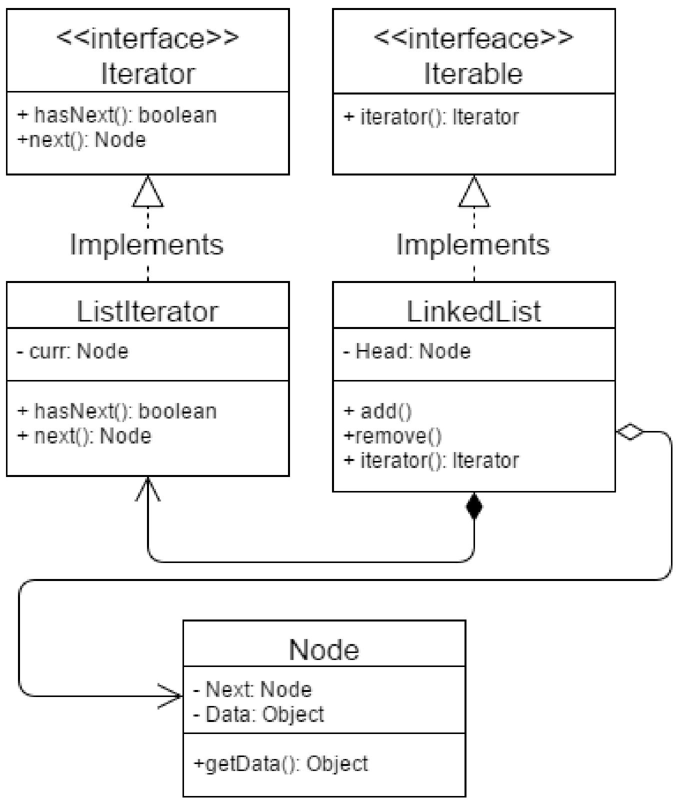
3. **(10 points)** How are the Adapter and Façade patterns similar? How do they differ? Be **<u>very</u>** specific to ensure full credit.

The adapter pattern is basically to provide a means for the client to deal with some entity that they may not otherwise know how to deal with. The Façade pattern provides unified interface to deal with a complex sub-system of different interfaces or classes and the client does not need to know about the gritty details of said sub-system.
Both provide functionality so a client can deal with something they don't know about, or shouldn't know about. They both follow the idea of abstraction. They are different because adapter is just converting something unknown to something know to the client. With façade, it will most likely provide the client with a lot more functionality because it is dealing with an entire sub-system that may have a huge amount of behaviors. Also with adapter it contains an instance of the class it is dealing with and façade will not because, again, it is dealing with lots of lower level, complex systems. In other words, adapter and façade have the same goal, but adapter is smaller scale and façade is larger scale.

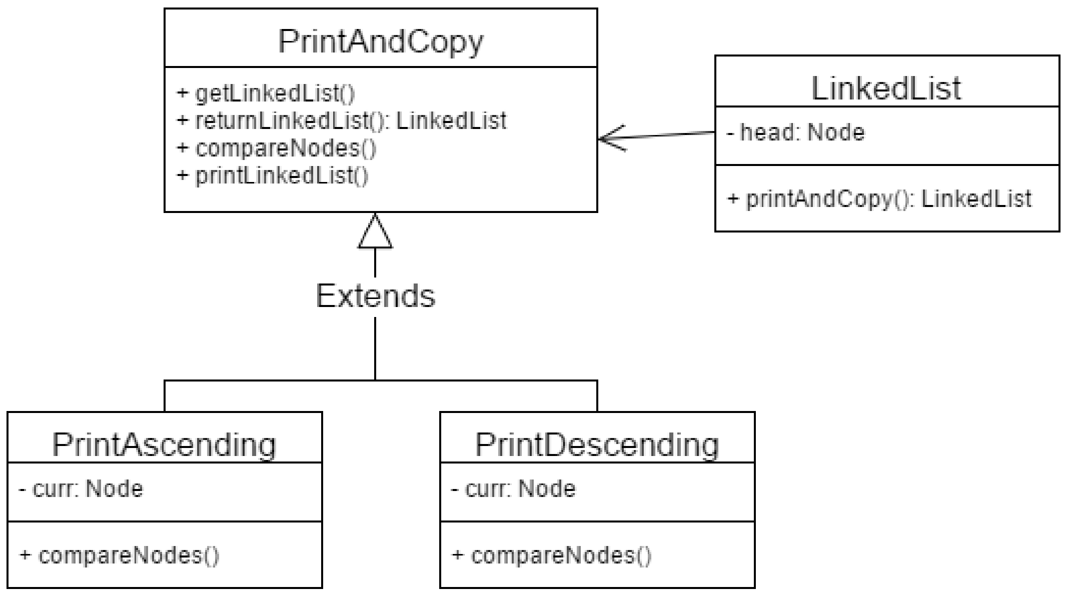
4. (**10 points**) (a) Write a class diagram for the **Template Method *and*** then (b) the **Iterator** design patterns. Note they are separate problems.

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.

Iterator:



Template:



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).

1. Single Responsibility: The notion of a given entity/class having one and only one function or responsibility in the given design, and with this needs to encapsulate the given functionality as whole. The idea is make each entity independent from the other entities in a given design. One pattern that follows this is the Factory Pattern.

2. Open/Closed Principle: The notion that a given entity is open for extension and closed for modification. This is basically the idea that something else is allowed to implement/extend the given entity but not able to modify its attributes or behaviors. MAKE SURE things are encapsulated in their entirety so nothing can be modified. One pattern that follows this is the decorator pattern.

3. Liskov Substitution Principle: The notion that a reference to some parent class of a class hierarchy can be substituted with a child class of that parent class. It is kind of similar to the open/close idea. That we can use an extension of a given entity and not change anything higher up in the hierarchy. “Super class reference to a sub-class object.”. One pattern that follows this the strategy pattern.

4. Interface Segregation: The notion that no given entity should have to know about things that it does not need to. In practice, we could say that if some interface that we are implementing has more behaviors than we need to use, then we should think about making smaller, more specialized, versions of the interface. One pattern that follows this is the adapter pattern.

5. Dependency Inversion: The notion that a higher-level entity is not working directly with a lower level/complex entity, but rather these higher-level entities are working through abstractions. It provides more decoupling of code and creates entities that are less complex to maybe accomplish something that is complex as a whole. One pattern that follows this is the template pattern.

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

    1. Comments: Always explain why you did something, not what you did. You can re-factor this by writing more intent revealing code, i.e. good variable names, good method names, etc.

    2. Duplicate code: Do not ever repeat yourself. A good programmer is a lazy one. This is basically writing code that you have already wrote to so the exact same thing. To re-factor, use abstractions, utilize template pattern.

    3. Conditional complexity: This is basically when you have huge logic blocks to accomplish something in your code. To re-factor this, we can always use composition, follow OO principles, also we can utilize the state pattern.

    4. Inconsistent names: This is basically when, in our code, we use different names for things that are similar. This is can cause confusion when people review your code. Re-factor this by sticking to a standard at which you name things within your code.

    5. Indecent exposure: This is the notion of making things that should be private, public. This is bad in so many ways, it goes against encapsulation, it can cause security issues, and can cause so much more issues as well. To re-factor this, we will just simply make attributes private, also things that only a certain package, hierarchy, etc. need to know about we can issue visibility as necessary.

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

    Coupling is basically the idea of how closely two entities are to one another. This also describes how dependent on other entities a certain entity is. The more coupled, the more dependent. Cohesion has to deal with what an entity is doing. More specifically, if an entity has low cohesion it is more concerned with doing what it is the given entity is supposed to do as a whole. If an entity has high cohesion than that entity will have a lot of function what might be doing things that does support the overall goal of the given entity. These two concepts can be deeply related and must be dealt with heavily in software design. It is important that we do not have coupling in our code and with this we need to ensure lower cohesion. It is important though, to note, that this is a rule of thumb, and there will always be instances of this not happening and being perfectly acceptable. Not everything is always black or white, so do note that there are exceptions.

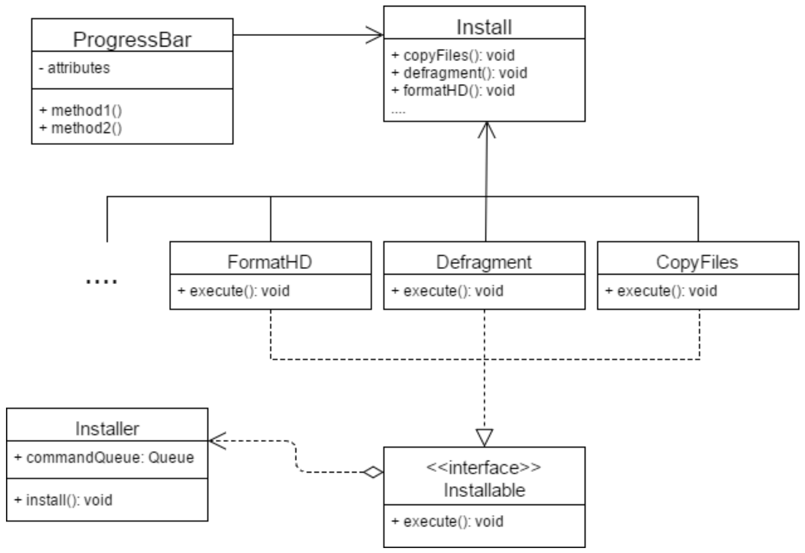
Design Problems

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.

For this example, we would definitely want to use the command pattern. This pattern is basically built for this function. We have a set of commands that need to be executed by the client in this case the installer. We can encapsulate each command into a class of its own so then we can issue the commands without knowing what they do.



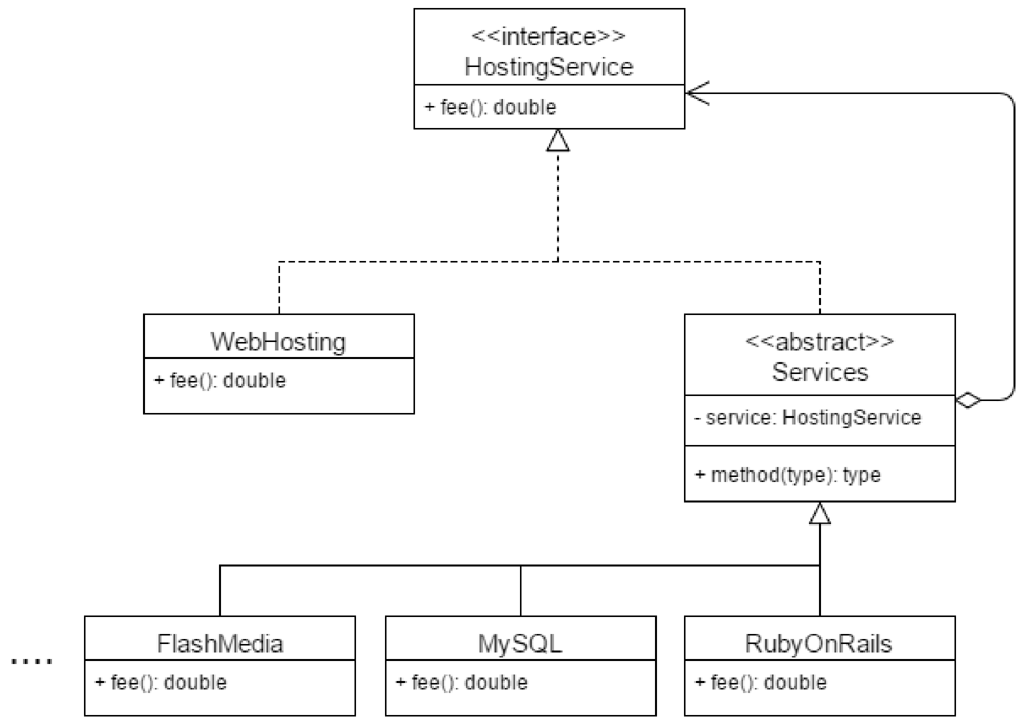
```
Install install  = new Install();
FormatHD fhd = new FormatHD();
……
Installer installer = new Installer();
installer.execute(fhd);
……
```

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.
```

For this example, we will want to use the decorator pattern. The decorator pattern would be the best option for this problem because we can wrap the base service over and over with different services and add up each cost to come up with the final cost.

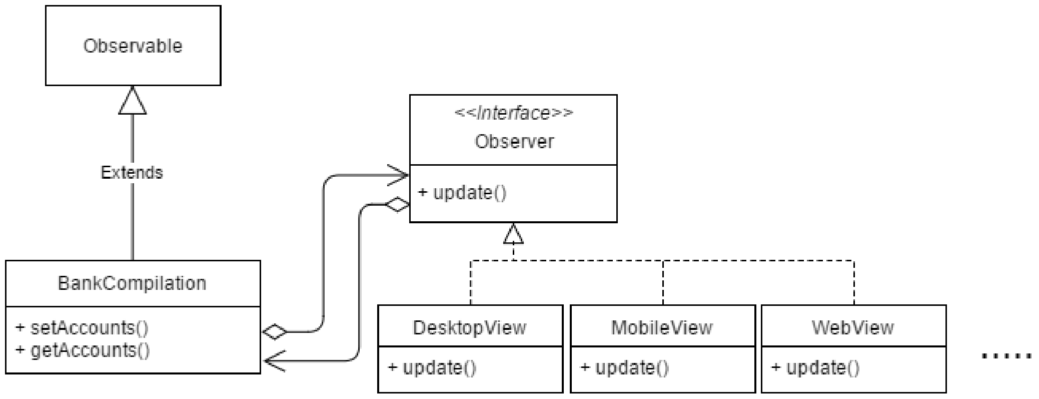


HostingService service = new WebHosting(new MySQL(new FlashMedia));
double totalPrice = service.fee();

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.

For this problem, I do think there is more than one way to go about solving it. For me I think the pattern that makes the most sense would have to be the observer pattern. To use the observer pattern, we would have to set up each viewport as an observer to the core application, which in this case is the bank account compilation part of the application. This entity will be the observable object at which each viewport will be able to observe. Once the core part of the app's state is changed, the others will be notified and then change accordingly. I think this would be the best approach because there is really only one entity that is doing what the application is supposed to do, and the rest of the components are only changing the way you view that information.



```
public update(){
        this.bankComp.getAccounts();
        …..
}
public BankCompilation{
        ArrayList views = new ArrayList();
        public static void main(String [] args){
                for(Observer: views){
                        update();
                        …..
                }
                …..
        }
}
```

The getAccounts() method will essentially get the state of the current bank compilation. Also when ever that state changes, the observers will be notified and change according to how they are supposed to change within their respected update() methods.

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.

Non-functional requirements in software design is essentially how a system will behave when faced with some change or task. With something that is a functional requirement it is just simply what the system/app will do. Non-functional requirements may include not limited to, scalability, reusability, time and space costs, security, maintainability, and much more. One pattern that could help with non-functional requirements could be, strategy pattern. This could help with scalability because it would be easy to implement another algorithm, which could then be swapped out at runtime from the client side. Another pattern that could help with non-functional requirements could be the command pattern. This could be a pattern that is implemented when writing some way to achieve the updating of your application. Say its outdated, then we can just execute the command on the client. With this we could also use the state pattern to check if, in fact, the application is in need of an update i.e. in an invalid state. This could help with the maintainability, security, and many other non-functional requirements. So many patterns can help with all non-functional requirements, and you could even say ALL of them do by nature. You can write software to accomplish extremely complex problems and have it work just fine, but it could be ugly and even render the software practically useless. Applying any pattern WILL help with these types of issues.

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).

After reading the article, what I understand is, with both of published and public, they may both have the same visibility, but for an entity that is published this means that the entity is an implementation, or internal to the package it lives in. Such entities are not exactly the best to program to because they can be broken at any time. We tend to want to always program to a published entity. To be published is to be something is intended to be programmed to and has nothing that could break if implemented by the client.

NOTE: To save time I am just writing down here. For EVERY problem, the answer was derived using my brain, as well as my in-class notes. These notes are a detailed combination of the slides, your writing on the board, and your speech given during lecture time. All of my notes are written during class as well as when I study them and add some extra comments or thoughts. No other resources were used during the time it took me to do the midterm.