Assignment 1

1) The first UNIX principle can be related to the ISP SOLID principle. With splitting up the files and processes the user can choose whether they want to save the software as a file or a process. It also shows the DIP since the processes would depend on being run from a file but the files do not have to depend on the processes. The second one is definitely a SRP since it is a single item to do a single task. One tool to do one task is the definition of SRP. The third UNIX principle has SRP while also being LSP. This one is SRP because each of these items have their own tasks that accomplish separate things. It is LSP due to the fact STDERR and STDOUT both print to the screen. This means that you could use STDOUT as the base class and STDERR could be the derived class as the user won't be able to tell the difference if it prints STDERR through STDOUT since they both print to the screen. The fourth design philosophy can show LSP and OCP while also including DIP. This one has multiple principles since it talks about combining tools. LSP in this case because you could combine tools such as STDOUT and STDERR as previously stated. OCP because you can use certain tools in other tools without needing to change the original tool. And then finally DIP due to the fact that certain tools do not need to depend on how the others work but these other tools may need to know how the original functions. Philosophy number five is an example of DIP. The plain text does not have a different look on different systems. But different systems need to know how the plain text is supposed to be shown. Thus the systems depend on the plain text but the plain text does not depend on the system. Philosophy six Is a good example of both ISP and LSP. It is ISP because in modern UNIX systems they can be used with either a GUI or a CLI which allows the user to choose the interface they want to use. It can also be LSP because the GUI could replace the CLI completely for the user and do the same functions. The GUI is a complete substitute for the CLI. The seventh philosophy uses a lot of the SOLID principles but I will pick two for now. It is LSP and DIP. It is LSP because it can be designed to have applications that can replace others to perform the same task completely. This being the CLI vs GUI example. It is also DIP because it can use different processes inside of a module without the module having to know how they fully work. Finally number eight is a great example of SRP. UNIX is wanting to build a single OS that provides the function of being an Operating system. It is not trying to build an OS with a bunch or rules and policies that cause the OS to do more than what its one task is.

2) This code violates the LSP by having the class B which inherits class A have a different version of the function m(). By this the function m() in B cannot replace the function in class A without it causing issues due to the missing code functionality. As LSP states: *Derived classes must be able to completely substitute for their base class.*

3) While yes this code will work. The issue is it breaks the LSP by the breaking the Design By Contract idea. With this Duck Type approach we mess with the public interface contract as we have a hidden dependency inside of aFancyHelperFunction. We do not check if this is existing before we pass it which breaks the preconditions. Thus our public interfaces are M1 and M2 but if we call

the aFancyMethod with a class that only contains m1 and m2 then we will receive a failure when we hit the aFancyHelperMethod without the user even realizing it. So, while yes python allows for this to be “Working” code the code is at high risk of failure based off the LSP being broken.

4) An example use of this virtual class is say I want to implement only order fries. Currently as Implemented I have to also implement the functions for orderBurger and orderCombo which would just be left empty or throw an error because I am not using them. This breaks ISP because I cannot pick and choose what I want without having to choose the others as well.

   It would be better to do something like:

```
Class OrderService : IOrderService {
        IOrderBurger& IBurger;
        IOrderFries& IFries;
        IOrderCombo& ICombo;
        OrderService(IOrderBurger& IBurger, IOrderFries& IFries, IOrderCombo&
ICombo)
          : IBurger(IBurger)
            IFries{IFries},
            ICombo(ICombo) {}
        void orderBurger(int quantity) override { Do Something Here}
        void orderFries(int fries) override { Do Something Here }
        void orderCombo(int quantity, int fries) { Do Something Here }
};
```

   This implements bad cohesion and coupling due to the fact that these could be split up into their own. For instance you could have an OrderFriesService and an OrderBurgers Service and a OrderCombosService. These can be their own implementation classes that way the OrderService class is not having to deal with all of these. OrderService class in this looks very similar to the god class concept we talked about due to the fact that it contains everything.

5) The first thing I see is the breaking of LSP. When looking at this code say I want to pay with Cash. You cannot just take cash as the café relys on the CreditCard

and the creditcard would not be able to function correctly if using cash. This leads into the next issue I see.

Cafe is only working in this since with the Credit Cards. But in a real scenario a Cafe would work with multiple different payment types. The Cafe needs to have a simple way to handle new payment types. This way the payment methods can be different but not need a whole rewrite of the actual cafe function if one changes. As it is now it breaks the OCP because the code is open to modification. We want it to be only open for extension.

The next issue I see is the buyCoffee function seems to handle both creating the coffee and dealing with the payment process. This breaks SRP due to it doing more than one single thing it would be better to split these up into different functions to maintain the single function with single responsibility.

The next issue I see Is a break of the ISP due to the fact that the Cafe is being forced to depend on the Credit Card function. This is a violation as the Cafe should not have access to things it is not going to use. And when we give the Cafe the full access to credit card by passing it in. We could be giving it more than it needs. We can mediate this by only passing in a interface of a payment method that only gives the function in buyCoffee access to the function of CreditCard that it needs. Without having to have it deal with other functions and data that it doesn't.

The fourth violation is in the same sense of the ISP being broken and we are breaking DIP. With the fact that Café is our high level module in this instance and it is reliant upon the lower level module CreditCard. Instead we should pass a payment interface that only exposes charge to the Café.