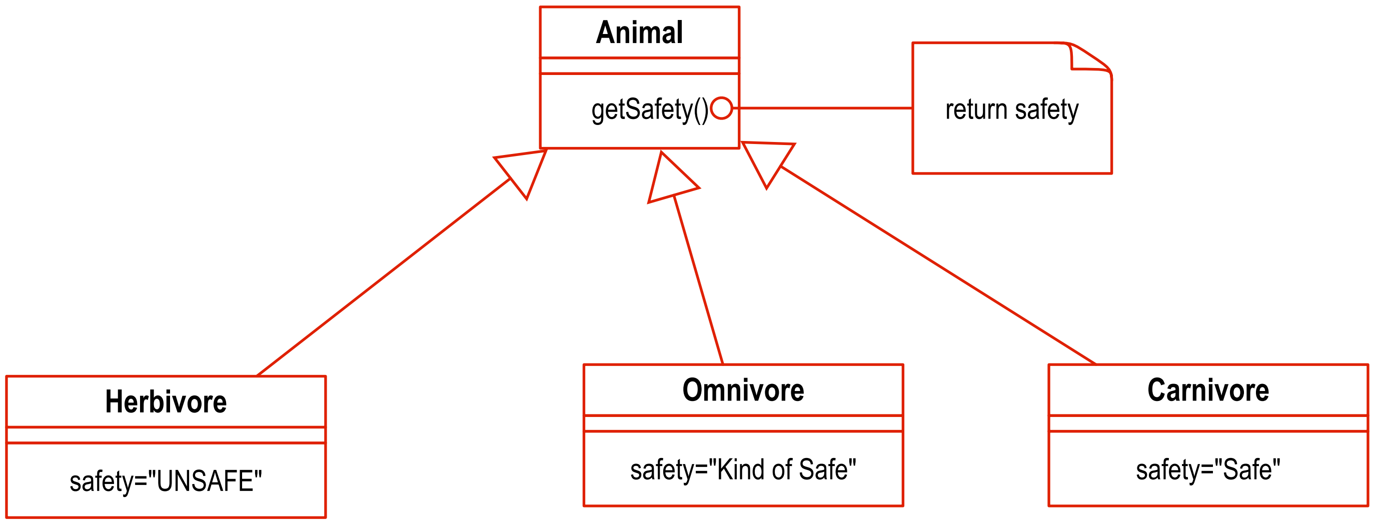
CPSC310 Sample Final Questions

1. What code smell is present in this method from the Zoo project's Animal class? Perform the refactoring to fix it, introducing any new classes needed, and changing the call-site as required.

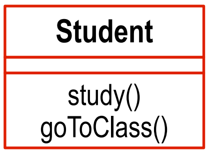
```
public class Zoo {
...
    public static void main(String ... args){

                        Herbivore
        animals.add(new ~~Animal~~("Koala"~~, "herbivore"~~));

                        Omnivore
        animals.add(new ~~Animal~~("Wolverine"~~, "omnivore"~~));

                        Carnivore
        animals.add(new ~~Animal~~("Dragon", ~~"carnivore"~~));

        ...
        for (Animal a : animals){
            a.getSafety();
        }
    }

}
```

```
public class Animal {
    ...
    String safety;
    ...
    public Animal(String name~~, String diet~~){
        this.name=name;
        safety="safe";
    }
    ...
    public String getSafety() {
        ~~if (diet.equals("carnivore")) {~~
            ~~safety="Safe";~~
        }
        ~~if (diet.equals("herbivore")) {~~
            ~~safety="UNSAFE!";~~
        }
        ~~if (diet.equals("omnivore")) {~~
            ~~safety="Kind of Safe";~~
        }
        System.out.println(this.name+" is "+safety);
        return safety;
    }
}
```



2. Provide a design example for each of the SOLID principles (meaning draw a small UML diagram or write some code and explain in your own words how that code/diagram satisfies the principle):

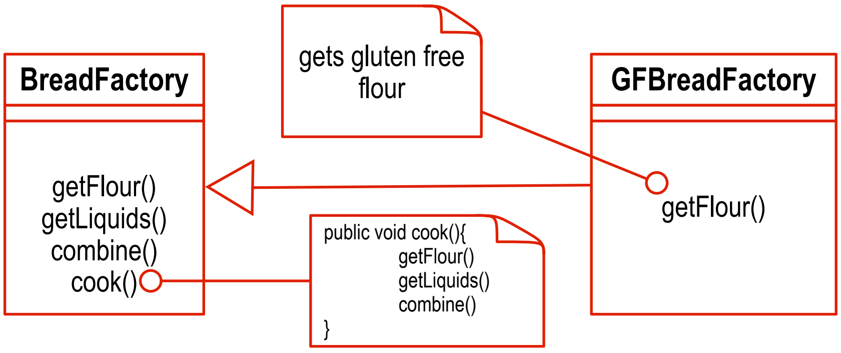
A. Single Responsibility Principle



The student only has studenty behaviour. (extra answer) if you looked at the fields that these two methods accessed you would see that it was the same fields.

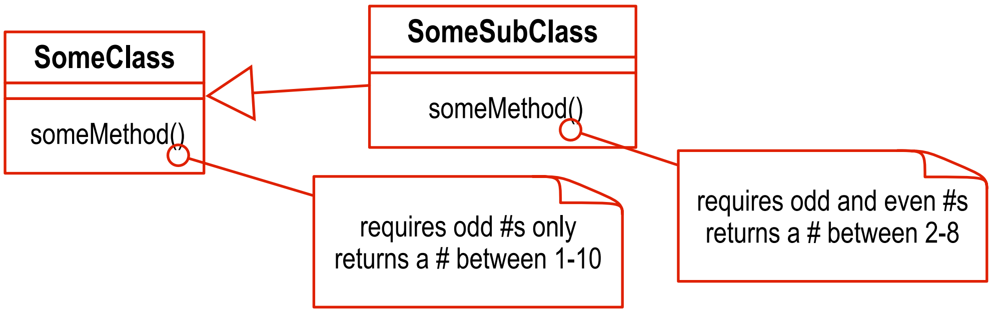
B. Open/Closed Principle

The BreadFactory has fine enough grained methods that the GFBreadBakery doesn't have to re-implement the whole cook method, it can just override the getFlour method to return the Gluten Free flour.



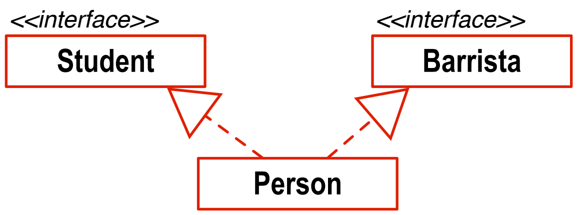
```
public void cook(){
        getFlour()
        getLiquids()
        combine()
}
```

C. Liskov Substitution Principle



SomeSubClass's implementation of someMethod: **weakens the precondition** and **strengthens the postcondition** of SomeClass's implementation of someMethod

D. Interface Segregation Principle



Each of these interfaces has a very specific role. The opposite of this might be a Person interface that combined all the possible Person traits.

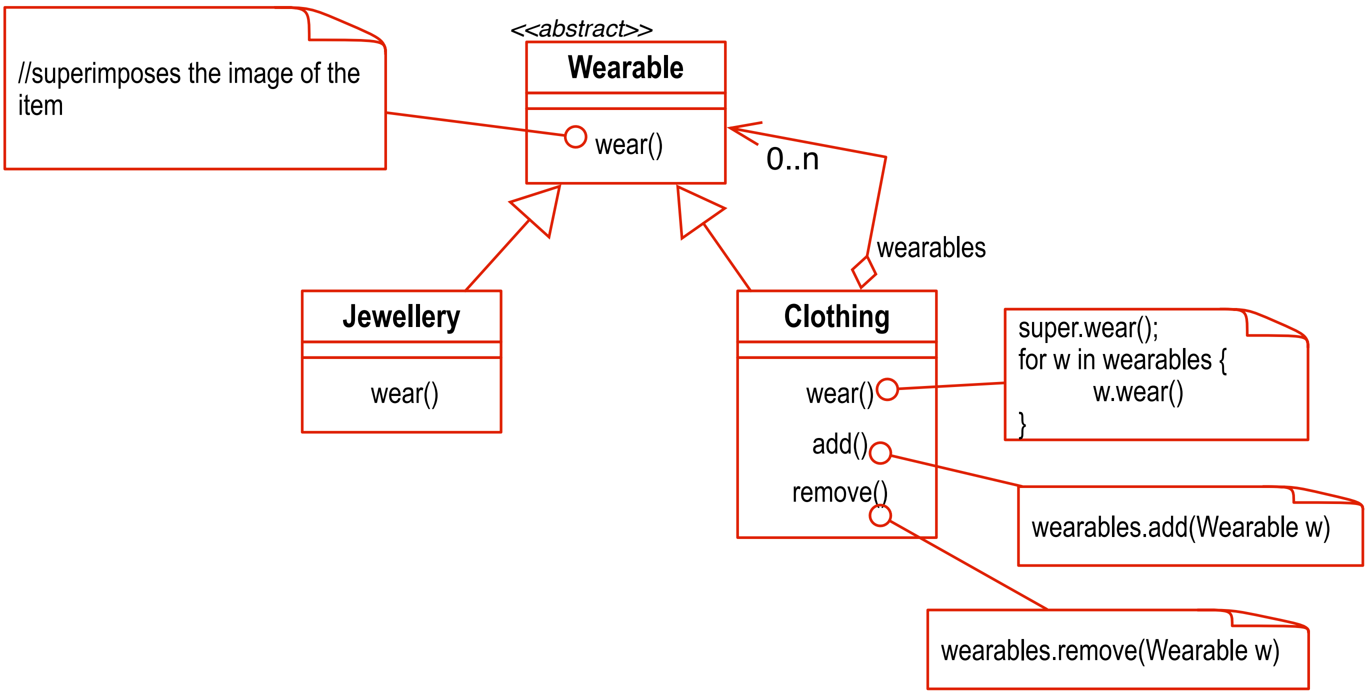
E. Dependency Inversion Principle

List myList = new ArrayList();

The class containing this line of code depends upon (the methods declared in) the List interface, not the ArrayList implementation.

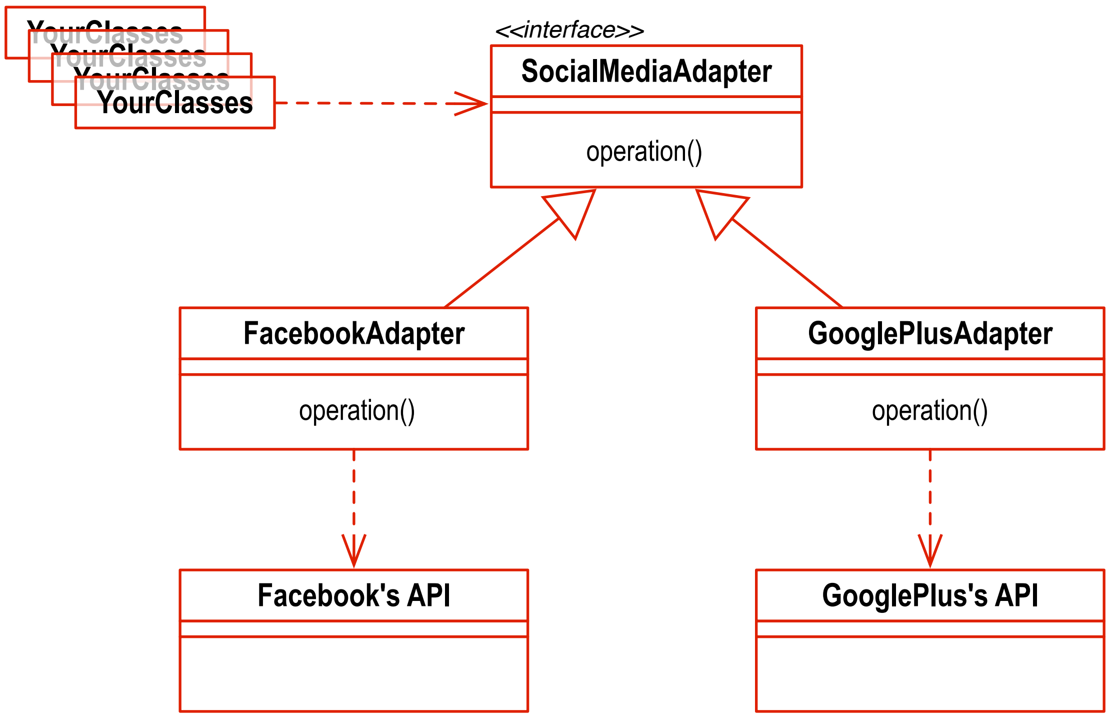
3. Use the Composite pattern to implement the **PrettyPrincess** application: clothing can be worn over other clothing, but jewellery cannot have other jewellery on top of it. Provide a UML diagram, and working code.

This is a slight allowable (and common) variation on the pattern where the operation method's default behaviour is put in the Component class because it is needed by both the Leaf and the Composite.

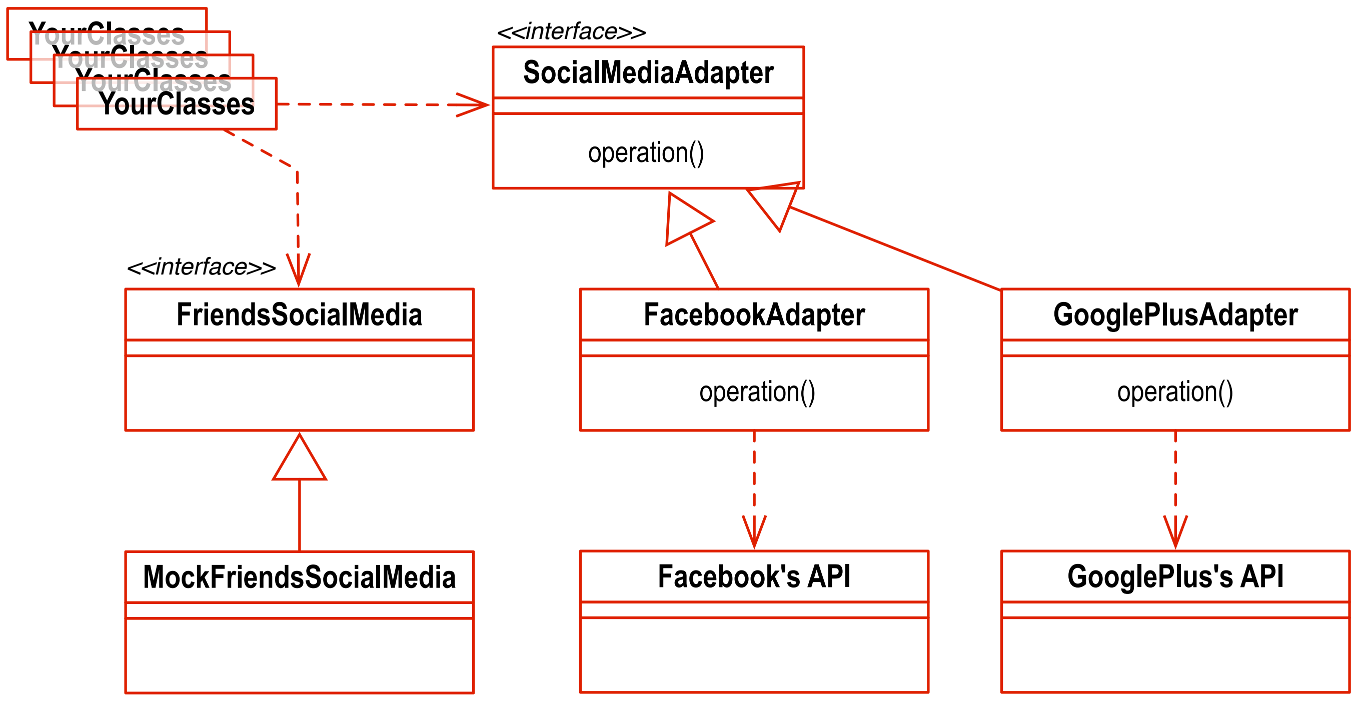


```
example main:

        Clothing c = new Clothing();
        c.add(new Jewellery());
        c.add(new Clothing());
        c.wear()
```

4. Assume you are implementing an application that makes use of two interfaces - the FaceBook interface, and the GooglePlus interface, depending on a user setting. Draw the UML diagrams and provide skeleton code for a solution using the Adapter pattern.

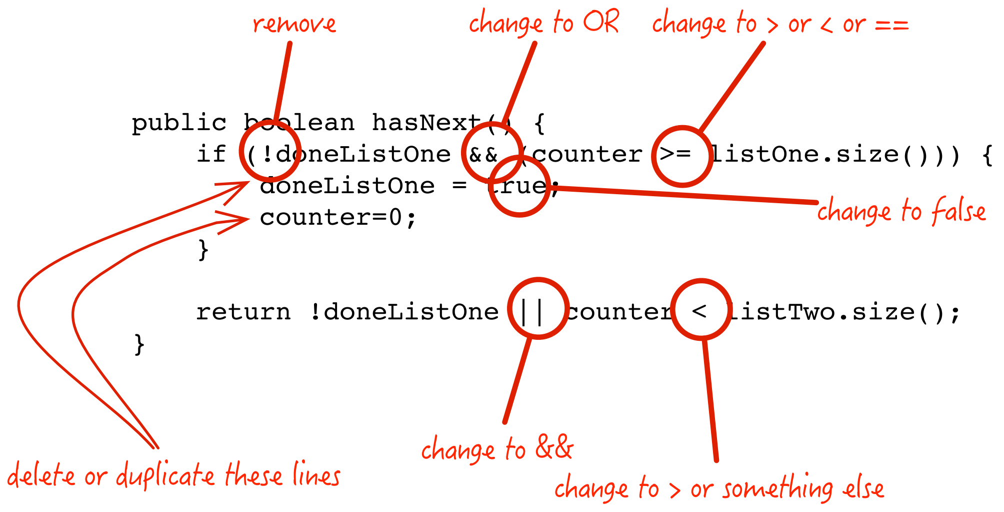


5. Now assume your friend is creating a third social networking interface UBCFriends, but is only partly done. Make a Mock Object to help you test your application while theirs is in development. Include configuration and instantiation code.



it's optional for the FriendsSocialMedia to implement the SocialMediaAdapter (it probably would, but nothing in the question requires that it does). The main thing in this answer is that the MockFriendsSocialMedia class has to extend the original FriendsSocialMedia class so that it is substitutable for it in the client code

6. Find 10 ways to mutate the following code:



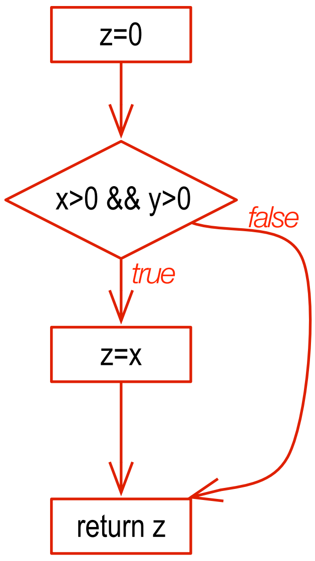
```
public boolean hasNext() {
    if (!doneListOne && (counter >= listOne.size())) {
        doneListOne = true;
        counter=0;
    }

    return !doneListOne || counter < listTwo.size();
}
```

7. Explain the difference between Basis Path coverage and Full Path coverage. Is one a subset of the other?

Basis Path coverage is a subset of Full Path Coverage
Full path coverage attains every combination of every condition's success/failure
Basis path coverage starts with all conditions being true or false, and each condition's success once.

8. Draw a CFG for the following method, and provide a test set that provides statement and branch coverage for this method:

```
int foo (int x, int y)
{
    int z = 0;
    if ((x>0) && (y>0))
    {
        z = x;
    }
    return z;
}
```



Statement coverage: {1,1}
Branch coverage: {{1,1},{0,1}}

Equivalence classes: x: -10,10; y: -10,10
Boundary tests: x: 0, -1, 1; y: 0, -1, 1

9. Imagine a system (Holiday Shopper) that covers the user story: “As someone planning my holiday shopping, I would like to be able to have a list of people to shop for and be able to add gift ideas under their names”. Write a user instruction to test a paper prototype of your application.

Hello user!
We have created an app that lets you plan your holiday shopping, such that you can have a list of people to shop for, and add gift ideas for each of them.
Please use this application to add a friend to your friends list, and add two gift ideas for that friend.
Please remember to think out loud while you work.
Thanks so much!

10. Imagine that the Holiday Shopper application is a client-server application, where all of the data is held on the server (to allow for updates from multiple clients).
   (a) Construct a RESTful API indicating, for each resource, the GET/PUT/POST/DELETE responses. Assume multiple users for this application
   (b) Indicate how Get/Put/Delete are idempotent
   (c) Indicate how the server is stateless
   (d) Indicate how the client does not need to recall the structure of the server side resources

RESOURCE	**GET**	**PUT**	**POST**	**DELETE**
users/	returns list of user IDs as links	not implemented	make a new user, add it to the list, return the user's ID	deletes entire list
users/userID	returns URI for friends list	not implemented	not implemented	deletes user
users/userID/friends	returns list of friendIDs	not implemented	makes a new friend, adds it to the list, returns the new friendID	deletes entire friends list
users/userID/friends/friendID/	returns the URI for the giftIdeas list	not implemented	not implemented	deletes whole friend
users/userID/friends/friendID/ giftIdeas	returns list of idea IDs.	not implemented	makes a new giftIdeaID, adds it to the list, returns giftIdeaID	deletes whole giftIdeas list
user/userID/friends/friendID/ giftIdeas/ideaID	returns the text stored at ideaID	replaces entire contents of ideaID	not implemented	deletes ideaID

b) Get Put and Delete can be repeated without any cumulative effect. For instance, putting new contents to an idealD will replace the contents of the idealD, resulting in the same final state for idealD after every put request.

c) The server maintains no knowledge of what stage the client is in their process. It has no "next" request, for instance.

d) The server returns all links needed for the client to navigate around the current resource structure. There is no need for the client to remember how the resources are stored. For instance, starting from a user, the client can get a series of links back from the server to navigate down to particular gift ideas.