1. CONCEPT QUESTIONS (short answer) [10pts]

[2pts] Briefly explain: Why are prototypes in the spiral model typically discarded when moving on to the next development iteration?

Because risks are reassessed; Because the subsequent iteration is written in a totally different language

[2pts] Briefly explain: What kind of "courage" is suggested by the extreme programming/agile programming approaches?

The courage to throw things away; The courage to refactor; The courage to test aggressively.

[2pts] Briefly explain: Why is copying and pasting code problematic?

Because errors will be propagated to all clones; Because changing one clone means changing all the clones

[2pts] Briefly explain: Why is it that code smells are present in production code?

Because refactoring happens *after* code is tested, and typically after code is released; Refactoring right before a release is unrealistic and not necessarily a good idea because of deadlines; Changes to be made in the subsequent release are not yet known, so you wouldn't know what code smells to target; Refactoring *everything* is neither recommended nor possible.

[2pts] Briefly explain: Why do we not clean up/refactor every code smell?

Because we want to make sure the smell is bad enough; Because we aren't sure what changes we're making so don't know what to target and don't want to make unnecessary structural changes.

2. WRITING GOOD USER STORIES [10pts]

Write two user stories for a Twitter like application that allows users to post updates, and other users to follow certain users and receive their updates.

For each story:

- Include the specific role from which the story is written
- Include the goal of the user in the story
- Include the benefit of the user in the story
- Include up to 3 acceptance criteria/definitions of done, AKA testing criteria (we are treating these as the same thing)
- Do not include tasks.

Examples:
As a person with lots to say, I would like to be able to type something into an app, and hit “tweet!” and then have it appear on my feed.

- show that you can type something into the app
- show that the “tweet” button appears
- show that hitting the button results in the tweet appearing in the feed

As a person who is interested in what others have to say, I would like to be able to follow other people’s tweets by hitting a “follow” button on their screen and seeing in my updates view all the tweets they make.

- show that the “follow” button appears
- show that hitting it adds the tweeter’s tweets to the user’s updates view

3. TURNING USER STORIES INTO UML [10pts]

Consider this role/goal/benefit statement for the *LostAndFound* app:

> **Statement:** *As someone who is registered in a course, I would like an update added to my updates view when the professor posts my grade to the grading repository so that I can see my grade appear in the updates view*.

A: On the next page, provide a UML class diagram that satisfies the above role/goal/benefit statement. Do not include anything extra in your design. Only elements/behaviour needed to make the user story work should be included.

<u>For the class diagram</u>:

DO include *as needed*:

- classes
- interfaces
- extensions relationships
- implements relationships
- fields (and their types)
- methods (and their parameters and return types)
- associations with cardinality/multiplicity
- compositions/aggregations (don't worry about distinguishing between these, but do use them correctly for whole-part relationships only).
- depends upon (calls) relationships (*not needed if combining the two diagrams as we did in class*)

DO NOT include Java library classes (like those of type ArrayList) as individually drawn classes in your class diagram

B: On the next page, provide a UML Sequence Diagram that depicts the sequence of operations involved in the Role Goal Benefit statement above**.**

<u>For the sequence diagram</u>:

DO include *as needed*:

- objects, and their types and lifelines (fine to use class boxes if you're combining the diagrams as we did in class)
- method calls and parameter names
- duration bars for methods should be clear
- return names (types not needed)
- return arrows only if values are being returned (optional otherwise if it helps with layout)
- Java library classes (like ArrayList) if needed
- loops/conditional behaviour

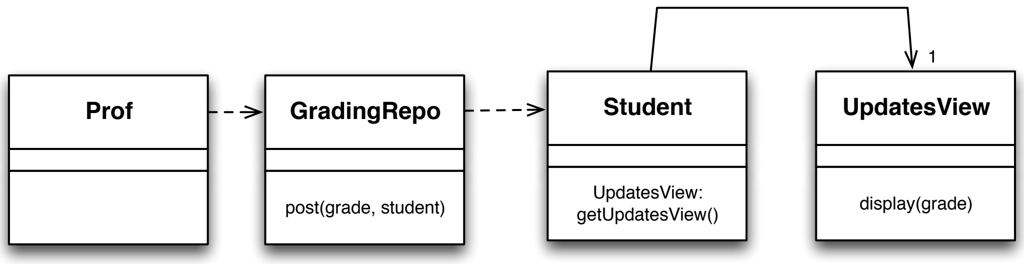
<u>Note: there is a UML cheat sheet at the end of the exam!</u>

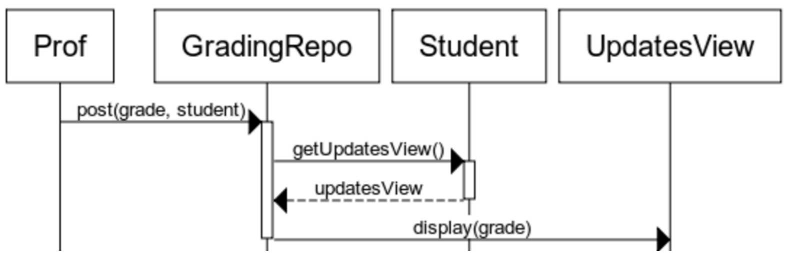
Remember to keep your diagrams free of the glaring errors we discussed in class.

NOTE: It is fine to combine the diagrams as we did in class, with class boxes along the top, and lifelines stemming down from them.

Answer Area for Question 3.

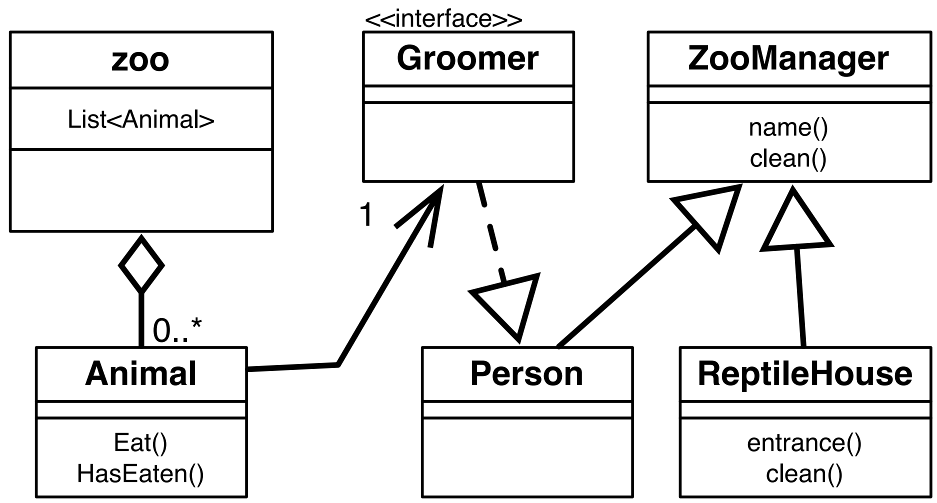
one potential answer:





4. FINDING GLARING ERRORS IN UML DIAGRAMS [10pts]

Recalling the glaring errors list from class, circle and label the violations you find the class diagram below. In the space below, expand on why you believe there to be a violation at that spot.
Note: there is a UML cheat sheet at the end of the exam!



LABEL	**Explanation**
2 points each up to 10	zoo should be upper case
	List<Animal> shouldn't appear — instead it should be an assoc link — it is not correct to say that it is already handled by the aggregation, because aggregations do not imply multiplicity.
	Eat() method should be eat(); AND HasEaten() should be hasEaten()
	Groomer implements Person is backwards OR Person isn't an interface so Groomer can't implement it.
	Person is not a ZooManager
	ReptileHouse is not a ZooManager
	entrance() should not be a method
	name() (if taken as a noun) should not be a method
-1 for each extra one.	There may be more correct ones additionally.

5. REFACTORING [10pts]

The following code allows developers to customise different styles of letters. Two letter-styles are implemented, Formal and Casual. There are two code smells in the code — fix them using the prescribed refactorings:

A. There is a problematic data clump passed into the constructors, that could lead to accidental parameter reversal. Apply the refactoring "Introduce Parameter Object" to address the data clump.

B. The developer has tried to reduce duplication in the printSignoff and printBody methods, but the printGreeting method still shows duplication between the FormalLetter and CasualLetter classes. Apply the refactoring "Move to Template Method" to fix that code smell.

Write the solution code on the subsequent pages (place code on each page indicated).

Rewrite the entire classes. If a method stays the same, write "stays the same".

```java
public abstract class Letter {
    String body;
    String toName;
    String fromName;

    public Letter(String to, String from){
        toName = to;
        fromName = from;
    }

    public void print(){
        printGreeting();
        printBody();
        printSignoff();
        System.out.println(fromName);
    }

    abstract void printGreeting();
    abstract void printSignoff();
    void printBody(){
        System.out.println(body);
    }
}

public class FormalLetter extends Letter {
    public FormalLetter(String to, String from){
        super(to,from);
        body="There has been a development of which you should be aware.";
    }

    @Override
    void printGreeting() {
        System.out.println("Dear, "+toName+",");
        System.out.println("I hope this letter finds you well");
    }

    @Override
    void printSignoff() {
        System.out.println("Sincerely,");
    }
}
```

```java
public class CasualLetter extends Letter {
    public CasualLetter(String to, String from){
        super(to,from);
        body="A whole bunch of stuff has happened";
    }

    @Override
    void printGreeting() {
        System.out.println("Hey, " + toName+",");
        System.out.println("How you doing?");
    }

    @Override
    void printSignoff() {
        System.out.println("Cheers,");
    }
}

public class Main {
    public static void main(String ... args){
        Letter letter = new FormalLetter("Menalaus", "Helen");
        letter.print();
    }
}
```

Enter your code for the Letter class here:

(one possible solution only)

```
public abstract class Letter {

   same fields.

   public Letter(ToFrom toFrom){
      toName = toFrom.getTo();
      fromName = toFrom.getFrom();
   }

   public void print(){
      printGreetingStart();
      System.out.println(toName + ",");
      printGreetingEnd();
      printBody();
      printSignoff();
      System.out.println(fromName);
   }


   protected abstract void printGreetingStart();
   protected abstract void printGreetingEnd();

   abstract void printSignoff();
   void printBody(){ same }
}
```

Write code for Formal Letter here:

```
public class FormalLetter extends Letter {
   public FormalLetter(ToFrom toFrom){
      super(toFrom);
      body="There has been a development of which you should be aware.";
   }

   @Override
   protected void printGreetingStart() {
      System.out.println("Dear, ");
   }

   @Override
   protected void printGreetingEnd() {
      System.out.println("I hope this letter finds you well");
   }

   @Override
   void printSignoff() { same   }
}
```

Write code for Casual Letter here:

```
public class CasualLetter extends Letter {
   public CasualLetter(ToFrom toFrom){
      super(toFrom);
      body="A whole bunch of stuff has happened";
   }

   @Override
   protected void printGreetingStart() {
      System.out.println("Hey, ");
   }

   @Override
   protected void printGreetingEnd() {
      System.out.println("How you doing?");
   }

   @Override
   void printSignoff() { same }
}
```

Write code for any additional classes here:

```
public class Main {
    public static void main(String ... args){
        Letter letter = new FormalLetter(new ToFrom("Menalaus", "Helen"));
        letter.print();
    }
}
```

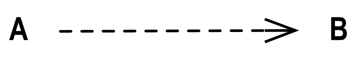
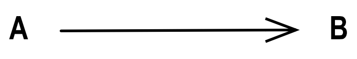
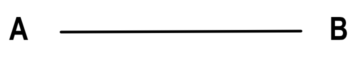
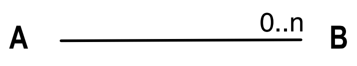
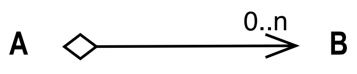
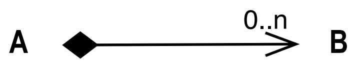
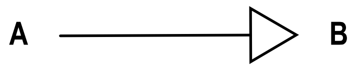
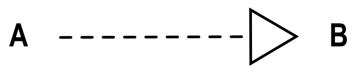
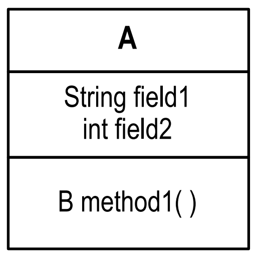
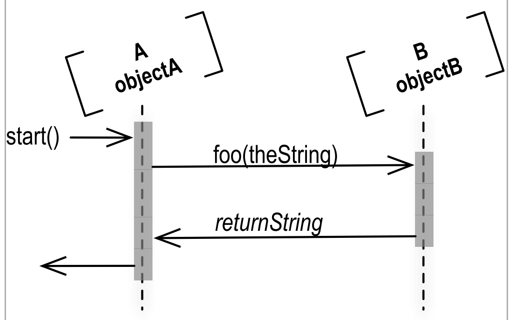
```
public class ToFrom {
    private final String to;
    private final String from;

    public ToFrom(String to, String from) {
        this.to = to;
        this.from = from;
    }

    public String getTo() {
        return to;
    }

    public String getFrom() {
        return from;
    }
}
```

UML Cheat Sheet (partial!) DO NOT REMOVE THIS SHEET OF PAPER!

UML	Name	Meaning
	Depends/ uses/calls	A depends upon a method or field in class B, uses class B, or calls a method in B.
	Uni-directional association	A has one field of type B (the default is 1, if no cardinality is listed)
	Bi-directional association	A has one field of type B, and B has one field of type A (default is 1 if no cardinality is listed)
	Bi-directional association	A has 0..n fields of type B's (probably a list of <B>) and B has one A (default is 1 if no cardinality is listed)
	Uni-directional composition	A is made up of 0..n parts of type B (probably a list of <B>). Those B-type parts will remain in existence if A disappears. Note: in this midterm we won't check the colour of the diamond
	Uni-directional aggregation	A is made up of 0..n parts of type B (probably a list of <B>. Those B-type parts will NOT remain in existence if A disappears. Note: in this midterm we won't check the colour of the diamond
	Subtype/ extends	A extends B
	Implements	A is implements B
	Class	A is a class A has 2 fields: `field1`, `field2` A has one method: `method1` `field1` is of type `String`, `field2` is of type `int`. `method1` has a return type of `B`, `method1` takes no parameters.
 A objectA B objectB start() foo(theString) returnString	Sequence Diagram	the method `start` is called on instance `objectA` of type `A`. `start` then calls `foo` on instance `objectB` of type `B`. the foo method has a parameter called "`theString`" which suggests that it is of type `String`. `foo` then returns "`returnString`" which, based on naming, is of type `String`. This ends the duration of the execution of `foo`. `start` returns `void` back to its caller. This ends the duration of the execution of `start`. The `void` return arrow is optional. Note: the call from `A.start()` to `B.foo()` would show up as a depends relationship in the class diagram from `A` to `B`.