MAHARISHI UNIVERSITY of MANAGEMENT
Engaging the Managing Intelligence of Nature
Computer Science Department
CS401 Modern Programming Practices (MPP)
Professor Joe Bruen

© 2015 Maharishi University of Management, Fairfield, Iowa

All rights reserved. No part of this slide presentation may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying or recording, or by any information storage and retrieval system, without permission in writing from Maharishi University of Management.

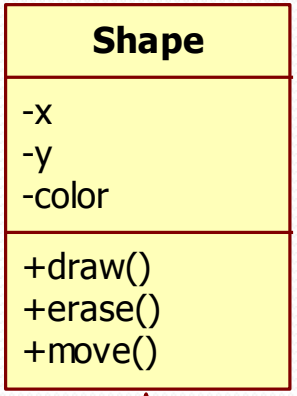
Lecture 4: Inheritance

Modeling “Is-a”

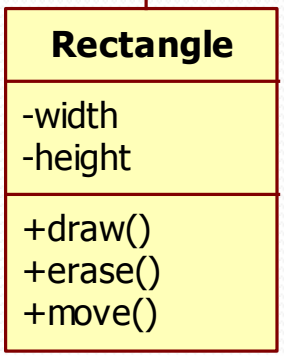
Inheritance

- Relationship between a general and a specific class
  - 'is-a' relationship
  - No multiplicity

(more general, abstract)
Super class
Base Class



(more specific, concrete)
Sub class
Derived Class




```
public class Shape {
    ...
}

public class Rectangle extends Shape {
    ...
}
```

Rectangle inherits all attributes and methods from Shape

Java Inheritance Review

- **`super`** object reference
  - To uncover parent methods, e.g. super.toString()

- **`super()`** constructor
  - To call parent constructor
  - Has to be first line in constructor
  - `Super()` will be inserted by the compiler into any constructor that doesn't explicitly use `this()` or `super() [with parameters]`

- **`final`** keyword on a class
  - prevents inheritance
- **`Protected`**
  - Allows access by subtypes **and** package
- **`Object`** The 'cosmic' super class
  - Provides methods such as `equals()`, and `toString()`

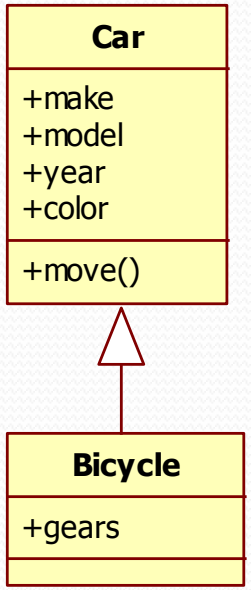
Overriding a method

- The subclass can change inherited behavior of the super class by overriding methods
- To override an inherited method, you must declare the method in the subclass in the exact same manner as the super-class method
- Best practice to also add the `@Override` annotation

```
@Override
public String toString() {
    return "Employee [salary=" + salary + ", getFirstname()="
    + getFirstname() + ", getLastname()=" + getLastname()
    + "]";
}
```

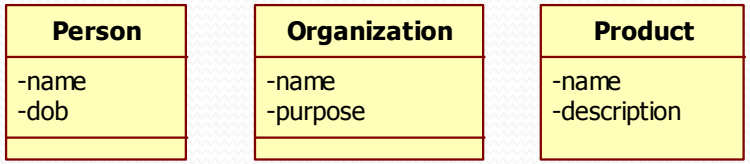
Code Reuse

- We've written the code for move() in our car class, and we want to re-use this code for our bicycle class.
- Why would the following diagram not be the best design decision?

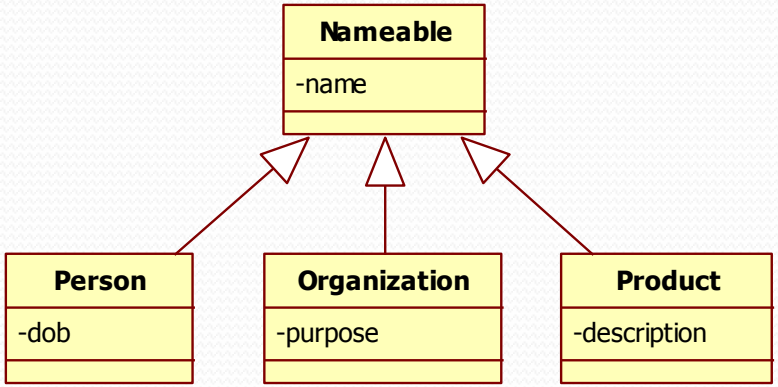


Code Reuse

- The following classes all have a name property



- Why may inheritance not be the best design decision?



- With design always think about the long term

Main Point 1

Inheritance is used to model ‘Is-a’ relationships.

Although Inheritance offers reuse (the subclass inherits all public and protected methods and attributes), reuse should never be your primary motivation.

The field of pure intelligence is inherited by everyone, and can easily be accessed through the practice of the TM technique.

Composition vs. Inheritance

Problems with Inheritance

When should I favor Composition over Inheritance?

Advantages of Inheritance

- Software re-use
  - Behavior inherited from another class does not need to be rewritten.
  - Increased reliability as the code will have been tested previously.
- Rapid prototyping, or incremental development
- Information hiding

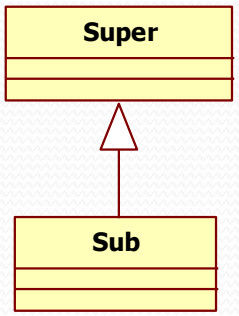
Problems with Inheritance

- Object classes are not self-contained.
  - Need to understand the super-class.
- Errors (and assumptions) in the super-class ripple down to the sub-classes.

…Composition also offers code re-use.

Inheritance vs. Composition

- It is easier to change the *whole* class of composition, than a super-class in inheritance.
- It is easier to add new subclasses (inheritance) than it is to add new *part* classes (composition), because inheritance comes with polymorphism.
- Delegation has a (small) performance cost compared to inheritance.



Inheritance vs. Composition conclusion

- Make sure inheritance models the 'is-a' relationship
- Make sure that this 'is-a' relationship is constant throughout the lifetime of the application.

- Don't use inheritance just to get code reuse.
  - If all you really want is to reuse code and there is no 'is-a' relationship in sight, use composition.

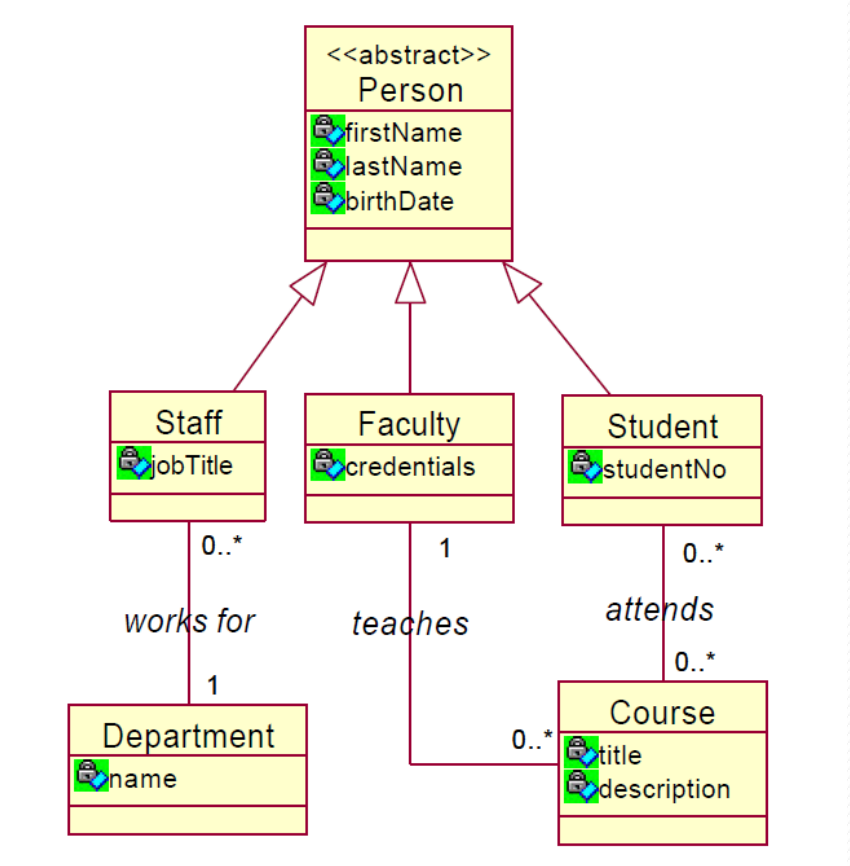
Main Point 2

Inheritance should only be used when you have a clear 'is-a' relationship, otherwise it is better to use composition beacause it has better support for change.

Even in clear 'is-a' relationships, Inheritance may not be the best choice due to its inflexibility.

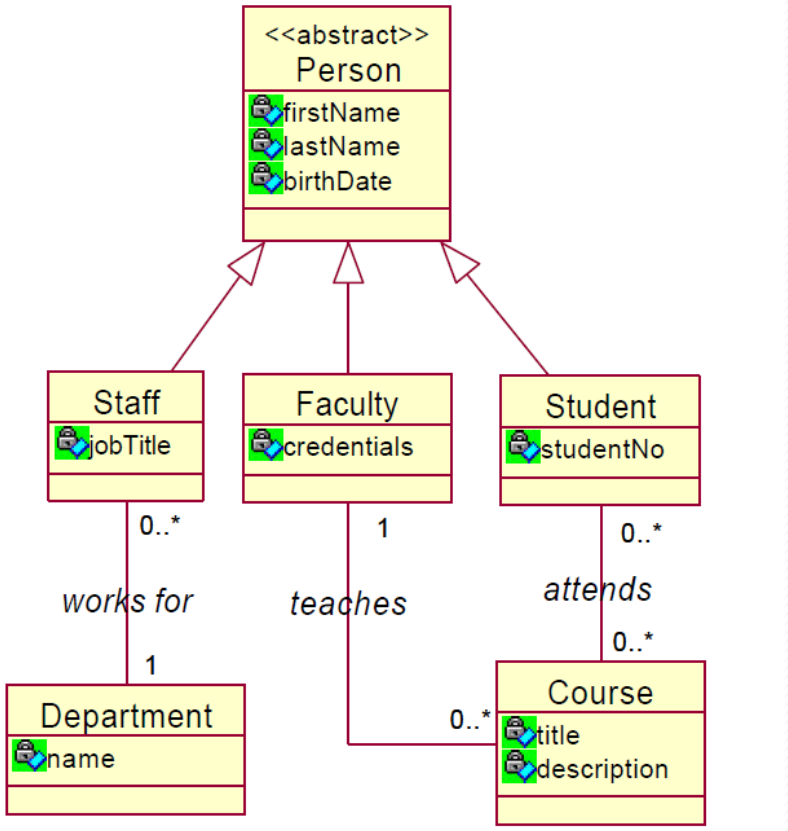
Software relationships that reflect the real world are more natural and easier to understand, thereby facilitating comprehension of the design by other developers (support of nature).

Example
Composition over Inheritance



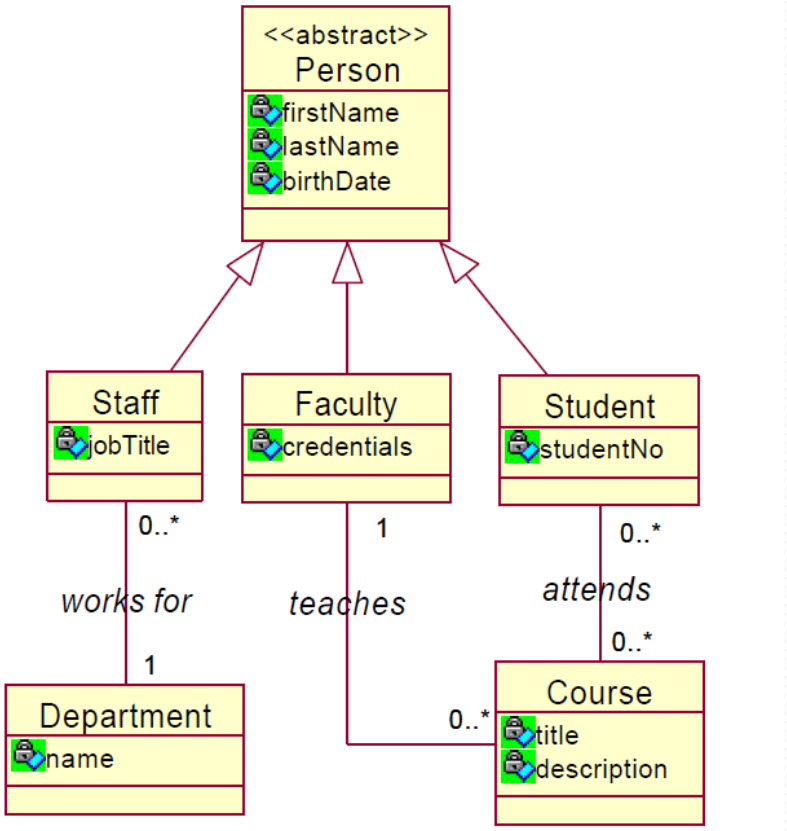
See any problems with this design?

Example
Composition over Inheritance



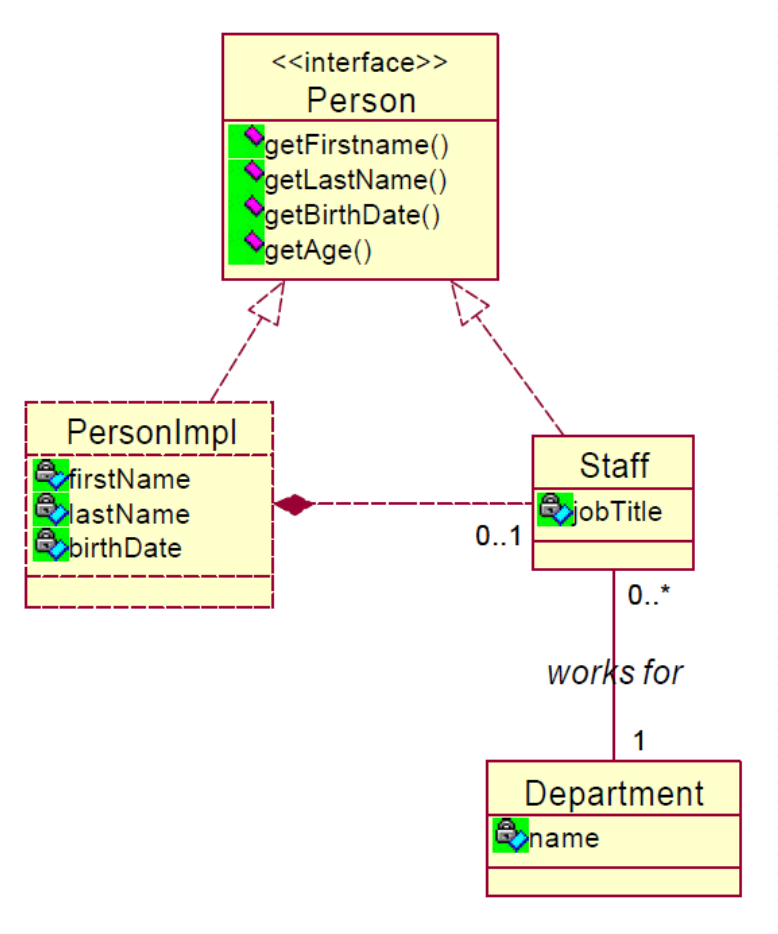
- Problems:
  - Inheritance is a static relationship and it must be decided at object construction time which type of person someone is
  - Once constructed a person cannot change from being a Student to being Staff or Faculty
    - In the **real world** people change all the time
  - Also a person cannot assume multiple roles of being a Staff member and a Student at the same time
    - Again, not how **it really works**

In-class exercise: Composition over Inheritance



- In your small groups try to redesign this class hierarchy using composition.

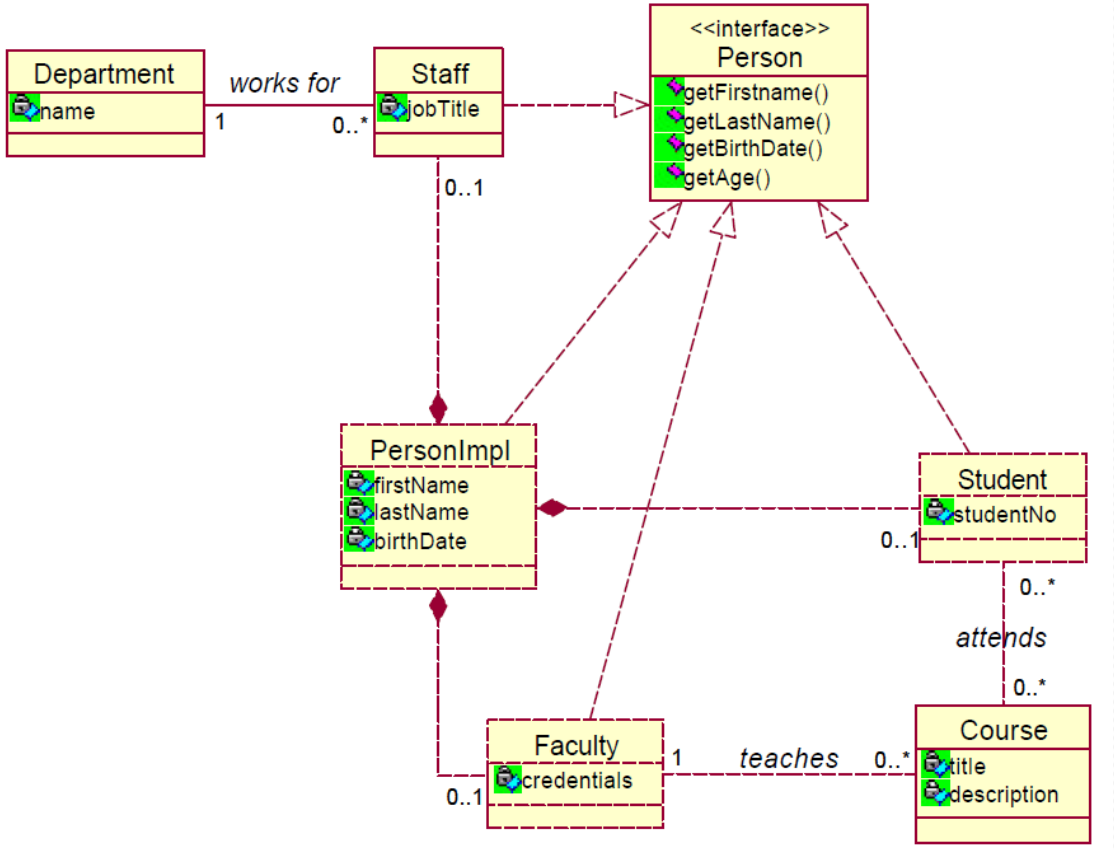
One Possibility



- Rename the `Person` class to `PersonImpl`
- Create a `Person` Interface that declares all public methods of `PersonImpl`
- Create composition between `PersonImpl` and `Staff`
- Implement the `Person` interface in `PersonImpl` and in `Staff`
  - In `PersonImpl` the `Person` methods will perform the actual business logic
  - In `Staff` the `Person` methods are delegation methods, i.e. they delegate the work to the corresponding methods in the PersonImpl object

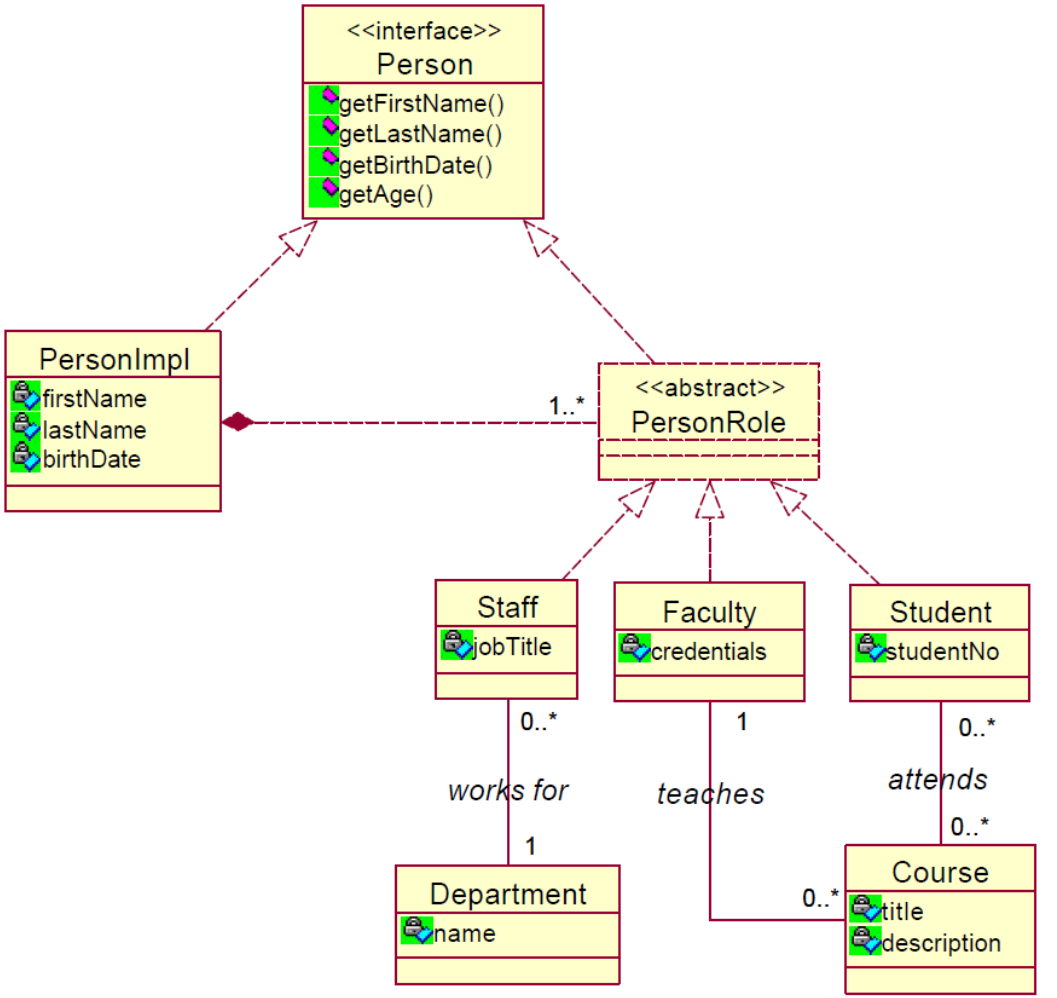
Problem

- If we do the same for our other classes the diagram will become cluttered (composition relationship must be implemented for each subclass)



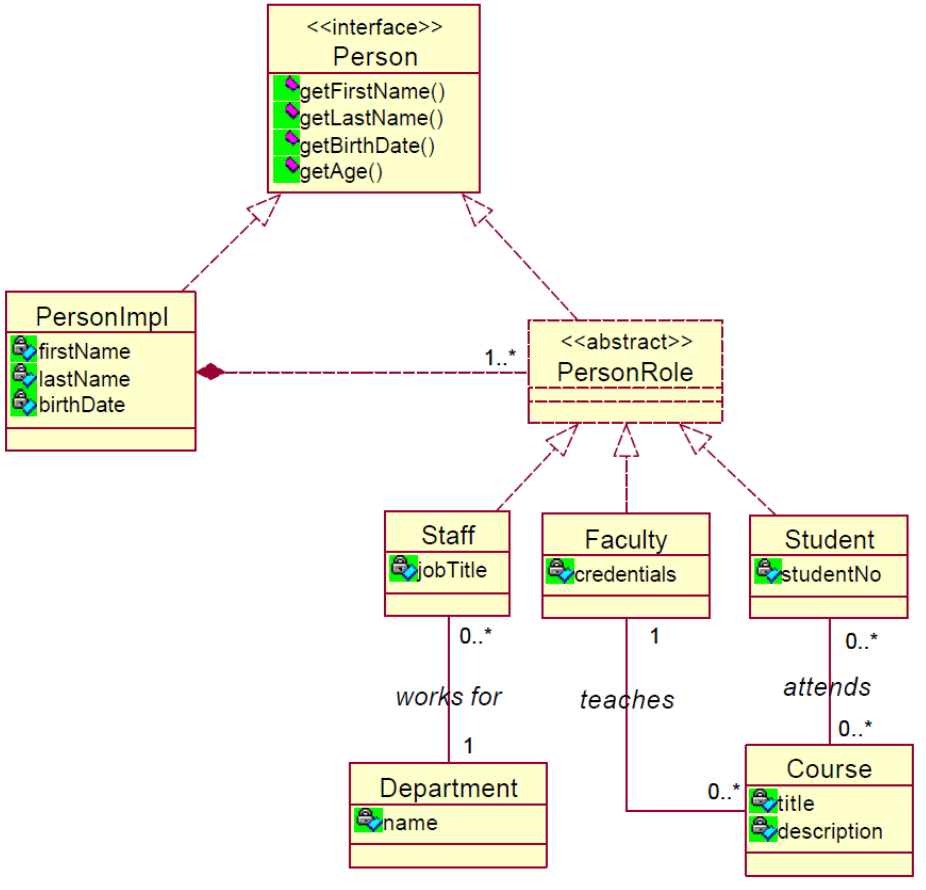
Solution

- We can fix this with a **`PersonRole`** super class



Solution

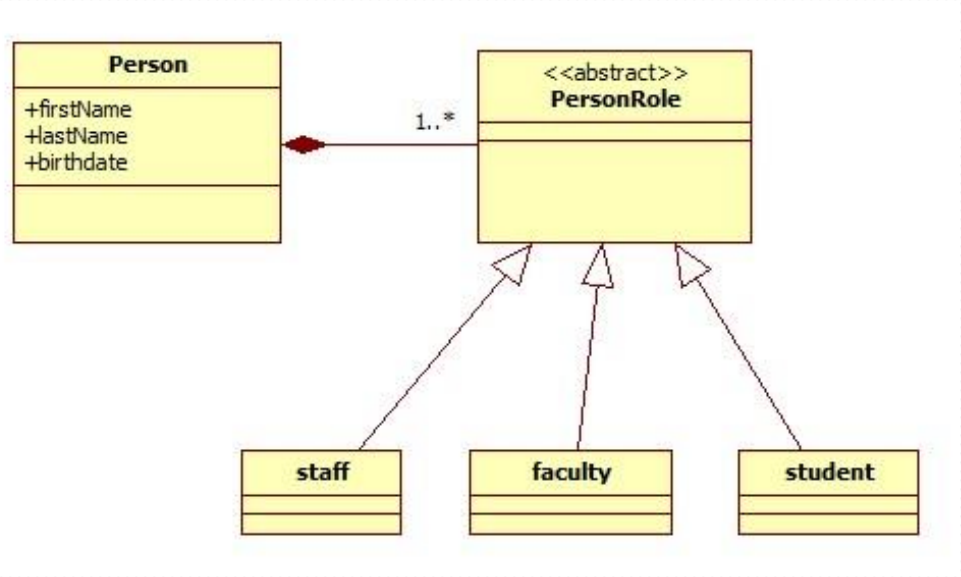
- Is there a reason for PersonRole to implement the Person interface??



Solution

- Why not make Person a concrete class and get rid of PersonImpl?

More on choosing Interfaces/Abstract Classes in Lesson 6.



Mid-term Practice

- Objectives:
  - Understand and discuss how to use Inheritance
- Tasks
  - Draw a UML class diagram for the following problem statement

Mid-term Practice

Problem Description: Our rent-a-wreck business rents cars, trucks, motorcycles, and mopeds. Create a inheritance model that we might use for our rentals.

1)Show a few common attributes and methods for your super-class.

2)Show some unique attributes and at least one unique method for each sub-class.

3)Show one method that will be overridden in all sub-classes.

Summary

Today we considered the pros and cons of using inheritance. We saw that we must be cautious when using inheritance because it is a permanent relation for the lifetime of an object.

Our goal is to build software that supports change and extensibility.

In general we know that composition has better support for change so we favor using composition except in cases where we have a clear 'is-a' relationship.

We see the same in life, at the surface level there is constant change, Problems arise when change is needed but not easily supported.

Connecting the Parts of Knowledge With the Wholeness of Knowledge

1. When requirements change, you should implement these changes by adding new code, not by changing old code that already works.

2. Inheritance and Composition are Object-Oriented principles that offer reuse of implementation.

---

3. **Transcendental Consciousness** is the field of pure intelligence that can be 'reused' by every individual at all places, at all times.

4. **Wholeness moving within itself**: in Unity Consciousness, the individual is united with everything else, and inherits the total potential of natural for fulfillment of all desires spontaneously.