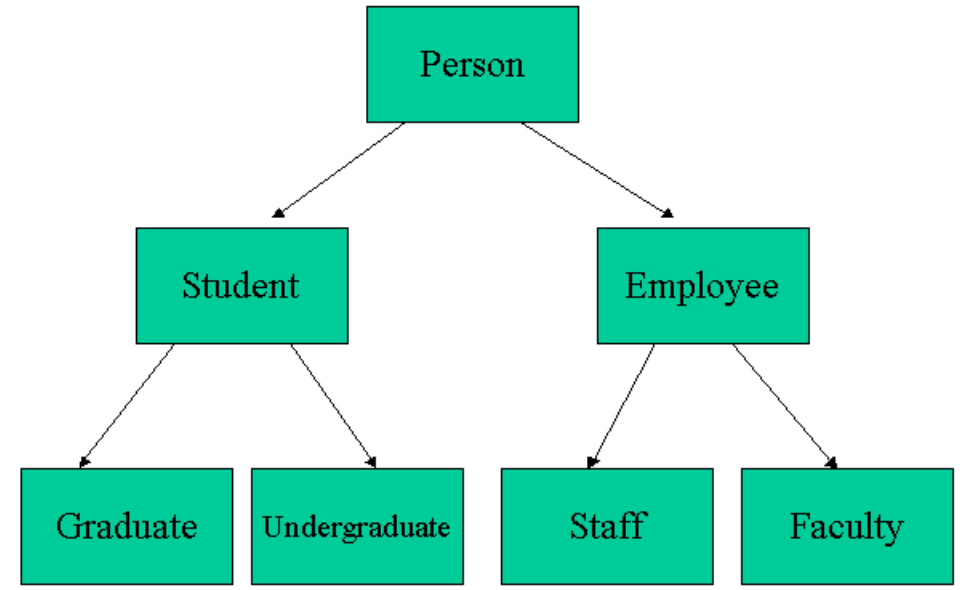
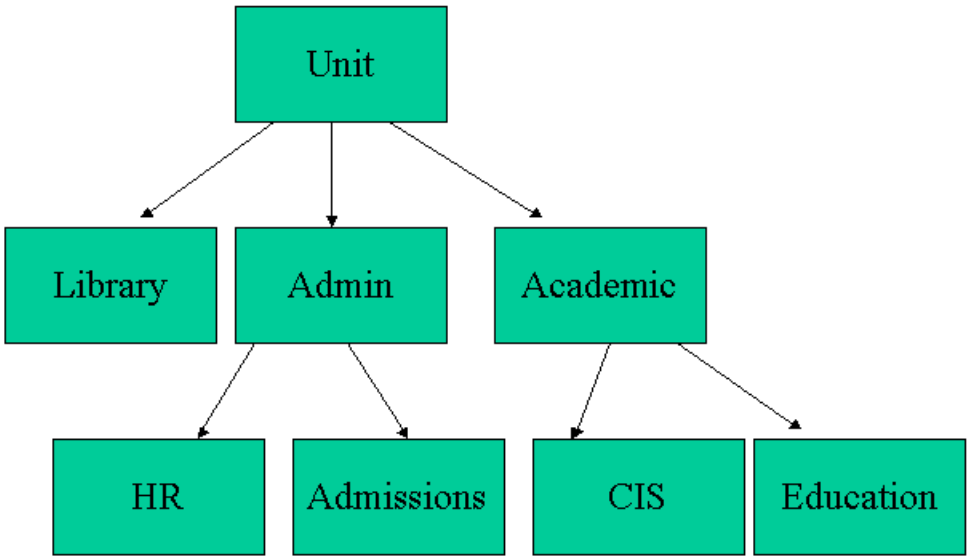
Homework 2 – Implementing the University Application

Due: November 7, 2004 (Sunday – midnight)

In this assignment, you will implement the University application from homework 1. Instead of implementing your proposed model, you will implement the classes following the inheritance trees below:





Every person on campus is affiliated to a unit. For example, a tenured faculty can be affiliated to the Education department.

In addition, a staff can be part-time or full-time, a faculty can be adjunct or tenure-track, an undergraduate can be in senior or junior, and a graduate student may be working on a masters thesis or PhD thesis. These will all be attributes of the classes.

The members of each unit (the leaves in the "unit" inheritance) are stored in a list of *Person* objects, where each *person* will be an instance of Undergrad, Graduate, Part-time staff, Full-time staff, Faculty, etc.. Despite this resulting in multiple "directories, a user of your system should not be aware of this, i.e., you should provide a "view" of a single University Directory, through a unified interface. The user can still ask, for example, for the list of all the members of a given department. However, this shouldn't mean that he/she has access to or has knowledge of the existence of a separate directory for each department.

The following sections will describe the main structure of each class:

1- The *UniversityDirectory* class

This class should have two constructors: one that initializes an empty directory, and one that creates a *UniversityDirectory* that has a deep copy of the existing *UniversityDirectory* instance passed as a parameter:

i) UniversityDirectory() {..}
ii) UniversityDirectory(UniversityDirectory source) {.. }

In addition, it should enable users to:

- Add/remove – Add/remove a *Person* by taking as parameter a variable of type *Person*
- The size of the directory.
- List all the contents of the directory, by calling the "toString" method of each element in the list
- Categorize the members of the directory, by finding out the number of undergraduate, graduate, adjunct faculty, etc. You shouldn't use any "type" attribute to find out the actual *person* in each position of the list.
- Search for a *person*, given a name or ID.

2- The *Person* class:

This class is the root of the inheritance tree. It is an abstract class that has abstract and non-abstract classes. The name of the methods are the following:

i) *getName()* – returns Person's name
ii) *getGender()* – returns Person's gender

iii) *getaffiliation()* – returns Person's affiliation (i.e., unit)
iv) *toString()* – returns a string describing the Person

You should decide which of the above methods should be abstract and which should be implemented in the *Person* class itself. You should also decide what attributes are information is used to implement these methods. You can change the parameters of the methods above, based on how you choose to implement them, e.g., if you override or overload them.

3- The subclasses:

Each subclass of *person*, whether it is an immediate child or a "grandchild", should:

- override at least one method from its parent
- define at least one new method
- define at least one method that is not (or cannot be) overridden by its child.

4- The "unit" tree:

The above requirements for *person* subclasses do not apply for the unit classes. Here, you can pick the attributes you want for each class. As, one of you pointed out, there are not many attributes you can think of to differentiate between "CIS" and "Education" departments, for example. However, I still want them as separate classes, since they will have separate directories.

Now, it is your decision to decide how each "unit directory", each subclass of "unit" and the "person" class are related. For example, you can have each "unit", e.g., the CIS department as one class, and its directory as a separate class. The fact that each "person" belongs to a unit can be modeled by the fact that a person is stored in the directory of that unit. Now, the question is how would you, for example, search a person in the directory? Would you just search for the person's name in all the unit directories? Or would you ask for the person's department, and then look in a specific unit's directory? In this case, how would you do the search, i.e., would you have a bunch of "if" statements that would compare the person's unit with all the existing unit direcotories?
Alternatively, would you have an instance of "unit" as an attribute of person? Or just a string with the name of the unit? Just remember all the patterns we covered and make your decision on that.

5- Other requirements:

Your design/implementation should include at least 1 case of:

- overloading
- composition

- substitution
- polymorphic variable
- deferred methods
- final methods
- true polymorphism

6- Based on all the items above and any other decisions you make, identify the design pattern(s) that best fit the requirements and describe how you implemented it.

Things to note:

i) I want a *toString* method in every class in the hierarchy tree. For example, the *toString()* method in the Adjunct Professor class should only return info directly encountered in the class. The value of attributes from the parent classes should be obtained by calling the parent's *toString* method.

ii) You will see that some of the classes in the figure above will not have much in it. That is OK, they are just being used to categorize the employees or students. They should, however, have a *toString* class describing what they are.

iii) You can choose the format for *address, ID, grades, etc.*.

iv) None of the classes above should have any "type" attribute. For example, the "Graduate student" class should not have a "type" attribute to indicate that it is an adjunct.

v) You can use either Java or C#

vi) I want a diagram of your entire system. You can use the UML modeling that we covered in class.

vii) Do not worry about implementing a variety of attributes for each class. Keep the classes small, and focus on applying all the concepts requested.

viii) Your grade will be based on whether you satisfy the above requirements, and how well you do it, in terms of you design choices, e.g., how well you reuse code, how efficient your code is, etc..

How to obtain data to populate the directory

Your main program should work in an interactive way, i.e., give options to the user and let the user choose/write the desired operation and the info needed to create a person. For example, a sample run of the program should be similar to the one below:

==========

C:\ java homework2

Welcome to the University Directory!! What do you want to do?

1- Add a Person

2- List all
3- List Categories
4- Number of entries (size)
5- List Unit
6- exit

Enter your option (1 to 5): 1

Adding a Person!! Enter some more info:

1- Employee
    a. Staff
    b. Faculty
2- Student
    a. Graduate
    b. Undergraduate

Enter your option : a

What is the name?
Gender?
Status?
Etc.. (whatever data is specific to staff)
.

Your entry was added successfully!!!

You want to continue?

Notice that I listed all the possible types of people to the user. You do not have to do it this way – you can choose any other way, as long as all the possibilities can be tested by your program.

Once you are done the implementation, answer how you implemented the following concepts in your program:

i) Polymorphic variable
ii) True polymorphism
iii) Composition (for code reuse)
iv) Subtyping
v) Inheritance
vi) Overriding
vii) Overloading (Scope or Type signature?)
viii) Coercion/conversion during substitution

ix) Copying (shallow or deep)
x) Abstract class and methods
xi) Identify the type(s) of inheritance used
xii) Default constructor
xiii) Refinement
xiv) Reverse polymorphism
xv) Design Pattern

Deliverable

1- **Well documented code for all your classes including the UML diagramas (35)**
2- **A page describing your design decisions, i.e., which methods were abstract and which were not, and why? How did you structure any additional classes you created. What design patterns you chose and how you implemented them. (45)**
3- **A short "user manual" describing how to use your program (5 points)**
4- **No sample runs, please.**
5- **Answers to the 15 questions above (15 points)**