THOMPSON RIVERS UNIVERSITY

Department of Engineering

Course #: SENG 3120

Course Name: Software Engineering Design: Process and Principles

Assignment 3: Object Oriented Programming Practice **Due Date**: March 16th, 2025

General Instructions

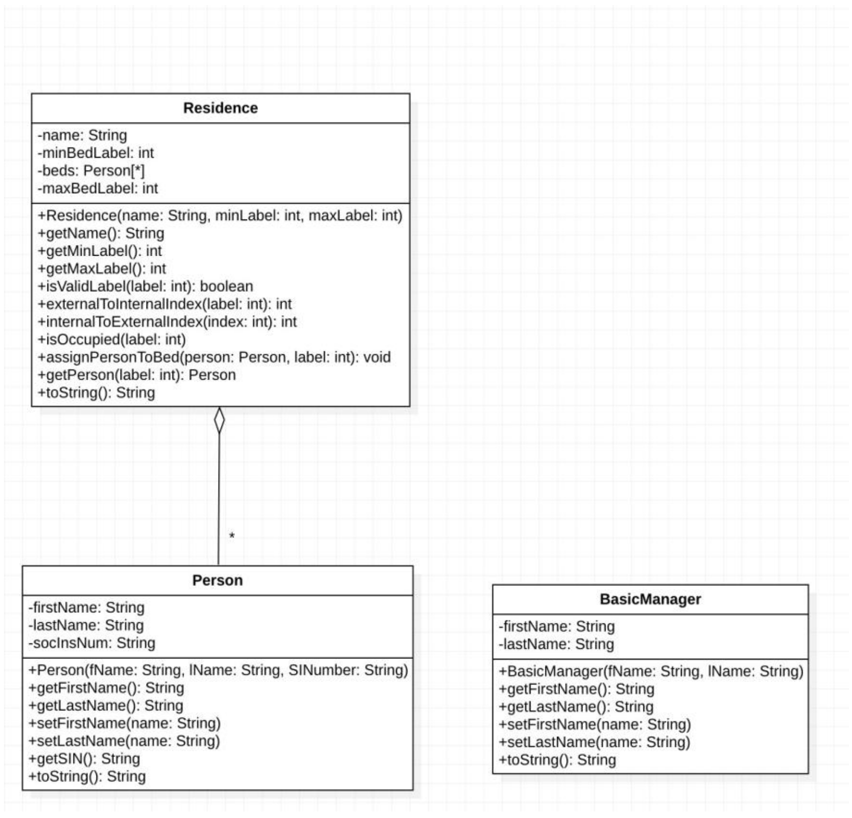
- **Read these instructions carefully.** You are responsible for all the instructions here.
- **This assignment is individual work.** You may discuss questions and problems with anyone, but the work you hand in for this assignment must be your own work.
- **Assignments are being checked for plagiarism.** We are using state-of-the-art software to compare every pair of student submissions.
- **Make sure your name and student number appear at the top of every document you hand in.** These conventions assist the markers in their work. Failure to follow these conventions will result in needless effort by the markers, and a deduction of grades for you.
- **Assignments must be submitted to Moodle.** It is your responsibility to ensure that your upload worked correctly.
- **The assignment is due at the time stated above.** There is a one-hour grace period until 12:59am, during which you work is not considered late.
- **Programming questions must be written in Java**. Do not submit the compiled .class files.
- **Instructions for submissions**
  - You will submit a single ZIP file containing all your work. You must submit a ZIP file, and no other archival format (e.g., no RAR, ?ZIP, etc.). If you use these other tools, and rename it with ZIP, the markers will not be able to open it, and you will get zero, because it could not be opened.
  - Each question will mention the name of the file(s) to submit, and file format requirements. Failure to follow file format requirements may result in severe mark deductions. If you change or modify any file you've previously submitted, you have to recreate the whole ZIP file, and resubmit it.

Overview

For this assignment, you will extend the work we did in the last deliverable of your project. We will be building a simple Student Residence Management System in a University with a fixed number of rooms/beds. The objective is to allow a user (some university administrator) to keep track of the students in the residence, and the managers of the residence. The users of this system will need to assign students to beds in the residence, and remove them from the system (say at the end of the year). Methods are needed to allow students to join and leave the residence. Also, it should be easy to determine which bed are empty so that a new student can be given an empty bed. As well, managers will be assigned to students (there will be multiple managers, and they may each have several students to consult with). It should be easy for a user of the system to see which students have which managers, and which managers have which students.

Assumptions

Engineers come up with deferent conceptual and architectural design alternatives. Let's assume that we started with the following design as shown in the following diagram. The goal of this assignment is to perform some improvement in the design and implement them in Java. I also provide Java implementation for this design, look at the files in A3_Starter.zip file. Try to understand the code in these java file as you are going to update the code there, perform restructuring and extend the code by providing more operations.


Residence
-name: String
-minBedLabel: int
-beds: Person[*]
-maxBedLabel: int
+Residence(name: String, minLabel: int, maxLabel: int)
+getName(): String
+getMinLabel(): int
+getMaxLabel(): int
+isValidLabel(label: int): boolean
+externalToInternalIndex(label: int): int
+internalToExternalIndex(index: int): int
+isOccupied(label: int)
+assignPersonToBed(person: Person, label: int): void
+getPerson(label: int): Person
+toString(): String
*
Person
-firstName: String
-lastName: String
-socInsNum: String
+Person(fName: String, lName: String, SINumber: String)
+getFirstName(): String
+getLastName(): String
+setFirstName(name: String)
+setLastName(name: String)
+getSIN(): String
+toString(): String
BasicManager
-firstName: String
-lastName: String
+BasicManager(fName: String, lName: String)
+getFirstName(): String
+getLastName(): String
+setFirstName(name: String)
+setLastName(name: String)
+toString(): String

General Requirements

Look the code provided, notice the naming conventions for Java classes, attributes and operation. Look how I wrote comments. This will help you to follow the same style. Each class should have a constructor, as well as the instance variables and methods specified in each question. All the instance variables should be declared private. You are expected to continue to **document your classes** internally using Javadoc comments, and other comments to make your code readable for markers. Proper internal documentation includes:

- A comment just before the class header that gives an overview of the class. For example, if the class models an entity, the comment might state what entity the class models and specify the key features. Alternatively, if the class serves as a container, state what type of items the container stores, and how these items are accessed. If the class is an interface, state what it provides an interface for and whether there is anything special about the interface. Finally, if it has a control function, what is it doing and controlling? Recall that comments for a class appear before the class and begin with /**.

- A Jacvadoc comment just before each instance variable stating what is stored in the field, again beginning with /**

- A comment before each constructor and method stating what it does. Note that the comment only gives what the method does, not how it does it. If it isn't obvious from the code how it accomplishes its goal, comments on how it is done belong in the body of the method.

- Be sure to include @param and @return comments. Also, if a method has a precondition, specify the precondition in a @precond comment, and throw a runtime exception if it is not satisfied. Note that a *precondition* in this context is an **extra** condition on the input arguments, e.g., a label can't be negative, or an STID has to be length 6. Note that these additional comments and precondition checks have already been added to the Person and BasicManager classes, but they should be added into the Residence class, as well as the new classes.

- Single line comments as needed to improve the readability of your code. Over-use of single-line comments will result in a deduction of marks.
- Use descriptive variable names and method names.

- Include good use of white space, especially reasonable indentation to improve the readability of your code.

Question 1: Practising Inheritance

In this question you will add a new class to the system, namely the Student class. This class extends the
Person class. For our purposes, a Student will have the following features:

- An integer label of the bed for the student. A value of -1 is used if the student has not assigned a bed
- A list to store all the student's managers (the Manager class is described in Question 2)
- A String to store a student's ID
- An integer to store a student's social security number
- Student(String name, int ssn, String SID) A constructor that takes in the person's name, social security number, and SID. Initially, the student should not be in a bed, and should have no manager assigned to them.
- getSID() An accessor method for the SID.
- getBedLabel() An accessor method for the bed label
- setBedLabel(int bedLabel) A mutator method for the bed label
- addManager(Manager m) A method that adds a new Manager to the list of the student's managers.

Hint: It's probably a good idea to create a stub method for this, until you have gotten a start on Question
2. You can come back and complete it when you have a better idea of how the Manager class will look.

- removeManager(String employeeId) A method that removes a Manager from the list of the student's managers
  Hint: It's probably a good idea to create a stub method for this, until you have gotten a start on Question
  2. You can come back and complete it when you have a better idea of how the Manager class will look. Study the BasicManager class, too, as the employee ID is an attribute there.
- hasManager(String employeeId) A method that checks to see if a Manager with employeeId is as- signed to the student. Returns true if the manager is found, false otherwise.
  Hint: It's probably a good idea to create a stub method for this, until you have gotten a start on Ques- tion 2. You can come back and complete it when you have a better idea of how the Manager class will look. Study the BasicManager class, too, as the employee ID is an attribute there.
- toString() A method that returns a string representation of all the information about the student in a form suitable for printing. This should include the student's name, social security number, the bed label (if any) and the name of each manager associated with the student.
- A main method that will test all of the above features

Hint: You can use the LinkedList or ArrayList class in the java.util library to store a list of managers. It's a generic class; review the lecture to remind yourself how to use these.

Question 2: Practising Inheritance

In this question you will add a new class to the system, namely the Manager class. This class extends the BasicManager class. Note that the BasicManager class also extends the Person class. Both the BasicManager and Person class are given on Moodle. For our purposes, a manager will have the following features:

- A list of the manager's students
- Manager(String name, int ssn, String employeeId) A constructor that takes in the manager's name, social security number, and employee ID (a string of digits, like a student number). Initially the manager should have no students.
- addStudent(Student s) A method that adds a student to the manager's list.
  Hint: You can make this a stub at first. You don't have to do everything in the order that the specifications are given. Once you have gotten attributes and constructors in place for Questions 1-3, you can come back to fill in some of the method stubs.
- removeStudent(String NSID) A method that removes a student from the manager's list.
  Hint: You can make this a stub at first. You don't have to do everything in the order that the specifications are given. Once you have gotten attributes and constructors in place for Questions 1-3, you can come back to fill in some of the method stubs.
- hasStudent(String NSID) A method that checks to see if a Student with NSID is under the manager's care. Returns true if the student is found, false otherwise.
  Hint: You can make this a stub at first. You don't have to do everything in the order that the specifications are given. Once you have gotten attributes and constructors in place for Questions 1-3, you can come back to fill in some of the method stubs.
- toString() A method that returns a string representation of all the information about the manager in a form suitable for printing
- A main method that will test all of the above features

You can use an ArrayList or LinkedList class (both from the java.util library) to store a list of students.

Question 3: Practising Inheritance

The Consultant class. This class extends the Manager class. For our purposes, a consultant will have the following features:

- Consultant(String name, int ssn, String employeeId) A constructor that takes in the Consultant’s name, social security number, and employee ID (a string of digits, like a student number). Initially the Consultant should have no students.
- toString() A method that returns a string representation of all the information about the manager in a form suitable for printing. This string should start off the the classifier "Consultant: " followed by the rest of the relevant information.
- A main method that will test all of the above features

Question 4: Maintenance

In this question, you will perform a maintenance task, by making modifications to the Residence class.

The **provided** Residence class should be modified so that the array has type Student, rather than Person. When this change is made, a number of the methods will need to be changed in order to be consistent with type Student being stored in the array.

In addition, the following two methods need to be added:

- availableBeds() A method that returns a list of the empty beds in the residence. This can be an ArrayList or a LinkedList.
- freeBed(int bedLabel) A method that removes a Student from a specific bed.

Question 5: System Design

The StudentResidenceSystem class. The last class to write is one to run a simple residence management system. This class has one purpose: to be the main program that allows the user to carry out the following tasks:

1. Quit
2. Add a new student to the system (no bed assignment yet)
3. Add a new manager to the system
4. Assign a manager to a student (this should also add the student into the manager's list of students)
5. Display the empty beds of the residence. See below for important note.
6. Assign a student a bed
7. Release a student from the residence. See below for important note.
8. Drop manager-student association (i.e., the student and the manager are dropped from each other's lists)
9. Display current system state (all students, all managers, all beds)

These tasks are numbered, and references to these tasks appear in the description below. Your system should display the above items to the console, and allow the user to enter a number from 1-9 to choose a task. Other parts of the system may require further input from the user, also through the console.

Our system can be implemented using the following instance variables (attributes):

- A single Residence object.
- A keyed dictionary of all the students known to the system, where the key is the student's SID. (Hint: Use a TreeMap.)
- A keyed dictionary of all the managers known to the system, where the key is the employee ID. (Hint: Use a TreeMap.)

The numbered tasks above will be implemented largely by implementing an instance method for each task (some, like task 1, don't need a method). Here are the methods we will need:

- StudentResidenceSystem() A constructor for the class. Initially, there should be no students and no managers. The Residence object needs to be created, and for that, the name of the residence and the integer labels for the first and last beds should be obtained from the user using console input and output.
- addStudent() A method that executes Task 2.
- addManager() A method that executes Task 3.
- assignManagerToStudent() A method that executes Task 4.
- assignBed() A method that executes Task 6.
- dropAssociation() A method that executes Task 8.
- systemState() A method that executes Task 9.
- toString() A method that returns a string representation of all the information about the StudentResidenceSystem in a form suitable for printing. For instance it might include information about the residence, a list of students and a list of managers.

- displayEmptyBeds() A method stub for Task 5. You do not have to implement this method; just create an empty method that could be completed in a future assignment.
- releaseStudent() A method stub for Task 7.
- A main method. The main method should construct a new instance of the StudentResidenceSystem class. Then it should go into a loop in which the user is presented a menu (see above), allowing the user to enter a choice to perform one of the tasks. The loop should terminate if the user asks to quit (task 1); before exiting, the system should display the current state of the system (as in task 9).

  Note for this assignment you do not have to implement task 5 and task 7, but there should be method stubs for these tasks that prevent the system from crashing if these tasks are selected.

Hint: You can use the TreeMap<K,V> class in the java.util library to store a keyed dictionary. Unlike the other classes, the StudentResidenceSystem class gets information from the user instead of having passing in arguments to each method. Every method above (except for toString() will require user input, even the constructor!)

Additional information

System Tasks: In each of the tasks, students are identified by their SID, managers by their employeeId, and beds by their (external) integer label. When the user quits, the system should print out the system state at that time. Note that when the task is to add a new manager, the user should be asked whether the new manager is a Consultant or not. If so, a Consultant should be created. Recall that, due to inheritance, an object of a certain type can be assigned to a variable of an ancestor type. Thus, a Consultant can be assigned to a Manager variable, and a Consultant can be placed in the dictionary of Managers.

What to Hand In

A zip file that contains

- All java files
- A separate pdf document that shows the UML class diagram for the system
- Be sure to include your name and ID at the top of all documents