SER515 Foundations of Software Engineering Assignment 2
Analysis and Design module, Agile, OO/Inheritance/DRY/Liskov code practice

The various files (Word docs, spreadsheets and zipfiles) asked for by each activity should be combined into one zipfile named ser515_<azurite>_Assign2.zip (no 7zip or other formats please).

PART I: Analysis Modeling (23 points)

1. Use Case Analysis (18)

Appendix A has an example use case I've used in other courses. In it, you will see the steps of developing use cases, preliminary and detailed, from a narrative. I've made several notes in *italics* throughout, this is for your study to assist with the project and NOT for this assignment task.

Your task is to construct an analysis model for these use cases. Specifically:

1. Construct an analysis class diagram showing the Actor, Boundary, Controller, and Entity analysis class and their relationships. This class diagram should encompass the 2 use cases for which use case specifications are given: UC-SG-1 Submit Grades and UC-RC-1 Register for Courses (only worry about the Step 4 version of this one).
2. Construct one activity diagram for the MAIN SUCCESS SCENARIO of use case UC-RC-1 Register for Courses (Step 4 only)
3. Construct a 2nd more detailed activity diagram for use case UC-RC-1 Register for Courses that includes the MAIN SUCCESS SCENARIO and also includes all success and failure variations in that detailed use case (Step 4 plus Step 5)
4. Construct a sequence diagram based on the use case UC-RC-1 Register for Courses (Step 4 only)
5. Answer: which ONE of the diagrams from 1, 2 or 4 (not 3) do you feel is the most useful if you had to pick only one, and why?

Submission: For this problem, please create a document named ser515_a2_part1_<asurite>.docx. Create image captures of your diagrams and be sure to answer #5. *I do not care what tool you use to create your diagrams, but you must use valid UML syntax. Our courses tend to use Astah, so you may want to use it now as you may need it in other classes. No hand-drawings please.*

2. Statecharts (5)

I am also teaching SER421 Web-based Applications online this Fall. Here is an edited excerpt from the current lab exercise:

- A *survey* is a collection of. A *survey* has the following possible states:
  A. *Created* – the *survey* exists.
  B. *Editing-In-Progress* (EIP) – A created survey transitions to this state once the first question is added to it. While in this state, questions may be added or removed while under the constraint that a survey may have N *survey items (questions)* where 0 <= N <= 5.
  C. *Completed* – authoring of the *survey* is done, meaning *survey instances* may be started from it. Surveys are considered completed when they reach the maximum number of allowed questions, or a completion event is given.
  D. *Verified* – once completed, a *survey* must be verified by external review and by automated tool review. If either review fails the survey returns to the *EIP* state. If both reviews pass, then the survey is considered verified.
  E. *Deleted* – *surveys* are not physically deleted in the system; they are only marked for deletion. This can be done at any time after creation. Once deleted the *survey* is immutable (can never become "un-deleted").

Create a UML Statechart capturing the semantics of the above description of a survey.

Submission: Add your statechart to your ser515_a2_part1_<asurite>.docx file. Again, I suggest a UML modeling tool, syntax matters

Part II: Agile/Scrum (17 points)

I've mentioned several times during our project discussions that in my SER516 class I have given a manual Scrum simulation assignment activity. This is a pared-down version of that activity, and is based on the popular Coffee-Scrum game (see handout).

a. Create an ordered Product Backlog of the User stories. You may do this by simply providing a sequence of ID numbers from the User Stories in the handout (blue numbers upper left corner). *This task is asking you to consider the tradeoff between Business Value and Storypoints (complexity for the dev team).* Briefly explain your prioritization strategy.

b. Simulate Sprints! You will simulate 3 sprints. For each sprint you have a capacity (velocity) of 10 story points

(1) Plan Sprint 1: From the stories tagged "Round 1" (upper right), decide which stories to put in Sprint 1.
   (i) The "round" simulates the Product Owner tagging these stories as the highest priority, yet you cannot fit them all in Sprint 1 (Round 1 stories add up to 23 points).
   (ii) Fill in the "Planned" column for Sprint 1 in the Activity spreadsheet with the selected stories (in order).
   (iii) Write a quick reflection on why you chose the stories you did. Don't worry, there are no "right" or "wrong" answers to the stories you select, as long as you adhere to the Rule.

(2) Execute Sprint 1:
   (i) Starting with the first story, roll a die. The die value is the *actual storypoints* required (meaning the actual velocity needed to complete the story). Keep track of this value by entering it in the sheet.
      1. If the value on the die is 6, move the Story to the "BLOCKED" column.
      2. Else if the value of the die is >= the planned Storypoints for that story, move the story to "Done". Note there is no "carryover" (if you have 2 remaining storypoints on the story but you roll a "5", then you record the 5 but you do not get to carry "3" over to the next story).
      3. Else (value must be < planned) move the story to the "In Progress" column, and update the remaining storypoints to be planned - value.
   (ii) *Go to the next story* and repeat step (b)(i). Note you cannot return to a BLOCKED story until all others are DONE.
      1. *For Sprint 1* the "next story" is literally the next story – the next one "down".
      2. *For Sprint 2* the "next story" is the same story if In-Progress; stick with it until it is DONE, then go down.
      3. *For Sprint 3* the "next story" is whatever In-Progress story you choose (same story, next story, some other one you choose) – but you must decide *before* you continue and roll the die again.
   (iii) The Sprint 1 simulation ends when either:
      1. The sum of the actual storypoints you are tracking on the side becomes >= the planned storypoints
      2. Or the stories have all been completed:
   (iv) After going through the stories in turn, if you have stories that are In Progress, you may return to them (in order) and repeat step b, adding the new die roll to previous die rolls to get a new actual storypoint value for that story. The completion criterion remains the same - if the actual >= to the planned for that story, move the story to complete. If you have stories that are BLOCKED but all others are DONE yet you still have actual < planned storypoints, roll the die to see if it can become unblocked. If you roll > 1 then move the story according to the rules.

(3) Complete Sprint 1: At this point either all of the stories you planned for Sprint 1 were Completed, or some are left that are In Progress or Blocked. Do the following:
   (i) Make sure the spreadsheet tab has properly recorded your sprint activity.
   (ii) Create a chart in Excel that has 3 types of burndowns embedded within it, visualizing the sprint just completed. Each increment on the X-axis is a roll of the die. The Y-axis is storypoints or business value, and should be initialized to the sum of the storypoints or business value of the planned stories in (a) above.
      1. The 1st BD should plot the reduction in remaining actual storypoints after each roll.
      2. The 2nd BD should only reduce by the planned storypoints when a story is Done after that roll.
      3. The 3rd BD visualizes the burndown of Business Value instead of story points. In this case, the reduction is the Business Value earned when a story is moved to DONE.
   (iii) Transition to the next Sprint (Sprint 1 and Sprint 2). All stories at the completion of the sprint that are in Blocked or In-Progress are put back on the Product Backlog and any prior reduction in actual storypoints is reset (go back to the storypoints on the card). Return to step a at the top and plan the next sprint, including these reset stories, and the stories for the next round (that is, if planning Sprint 2, then consider any "leftovers" from Sprint 1 and stories marked Round 2, if Sprint 3 you can consider all remaining story cards).

(4) Write a reflection on the project. Did it go well? Compare the 3 burndown charts across Sprints – which one do you gives the best information about what went on in the project? Compare the 3 selection strategies used in the 3 sprints as specified in b.(2).(i) – which one gives the best outcome and why?

Submission: Save the entire recording used in your spreadsheet to ser515_a2_part1_<asurite>.xslx. You may also enter your reflections in the spreadsheet on the first tab, ***Make sure you are using the spreadsheet to record your Sprints!***

Part III: Coding – Inheritance, DRY, Liskov, etc. (30 points)

This section is based on the coding exercises we have done related to Inheritance, DRY, and Liskov through class on 9/21.

IMPORTANT: Please follow the instructions given in each section for exactly how to name your source code files. We will be using automated test scripts as part of our grading, and if your failure to following the instructions here requires us to manually rename your files, we will automatically deduct 5 points!!!

1. Polymorphism, DRY, and Inheritance by State (15)

Download the code in LabPart3_1_DriverGiven.java. Try to compile the code, you will see it fails. With this code, do the following:

A. *Modifying code in the main method only*, make it so this code compiles and runs. The output should look like this (the actual values will be different due to the use of the random number generator.

```
THE EMPLOYEES
PartTime - Name: PTE1 Dept: 0 Wage: $28,583.96
FullTime - Name: FTE2 Dept: 2 Wage: $96,507.56 Options: 0
FullTime - Name: FTE3 Dept: 1 Wage: $15,388.36 Options: 0
PartTime - Name: PTE4 Dept: 2 Wage: $20,925.71
FullTime - Name: FTE5 Dept: 0 Wage: $68,371.26 Options: 0
THE EMPLOYEES WITH INCREASED OPTIONS
FullTime - Name: FTE2 Dept: 2 Wage: $96,507.56 Options: 90
FullTime - Name: FTE3 Dept: 1 Wage: $15,388.36 Options: 73
FullTime - Name: FTE5 Dept: 0 Wage: $68,371.26 Options: 10
THE EMPLOYEES WITH BONUSES
PartTime - Name: PTE1 Dept: 0 Wage: $28,583.96 bonus: $285.84
FullTime - Name: FTE2 Dept: 2 Wage: $96,507.56 Options: 90 bonus: $11,895.23
FullTime - Name: FTE3 Dept: 1 Wage: $15,388.36 Options: 73 bonus: $7,761.65
PartTime - Name: PTE4 Dept: 2 Wage: $20,925.71 bonus: $209.26
FullTime - Name: FTE5 Dept: 0 Wage: $68,371.26 Options: 10 bonus: $3,051.14
THE EMPLOYEES AFTER RAISES
PartTime - Name: PTE1 Dept: 0 Wage: $32,873.79
PartTime - Name: PTE4 Dept: 2 Wage: $23,221.85
FINAL STATE OF THE EMPLOYEES
PartTime - Name: PTE1 Dept: 0 Wage: $32,873.79
FullTime - Name: FTE2 Dept: 2 Wage: $96,507.56 Options: 90
FullTime - Name: FTE3 Dept: 1 Wage: $15,388.36 Options: 73
PartTime - Name: PTE4 Dept: 2 Wage: $23,221.85
FullTime - Name: FTE5 Dept: 0 Wage: $68,371.26 Options: 10
```

For submission, name this file LabPart3_1_DriverA.java. Of course, you need to change the class name to match!

B. *Apply inheritance by state* to create new solution that maximizes reuse and minimizes the way in which you had to solve part A (if you solved part A correctly you will know what this means). The output should look the same.

*For submission, name this file LabPart3_1_DriverB.java. Further, rename all *Employee classes to *EmployeeB to avoid conflicts with previous versions.*

C. *Add a new Employee type named HourlyEmployeeC* that has the following properties and behaviors:

   a. Instead of a salary, the HourlyEmployeeC has a payrate per hour. For example, $20 per hour.
   b. The HourlyEmployeeC also has a standard number of hours s/he works per week, in the range 5 to 40, which is set when the employee is created.
   c. A *raise* for an HourlyEmployeeC is computed as an increased percentage of the hourly rate. The percentage increase is accepted as an input parameter to the behavior.
   d. HourlyEmployeeC does not receive a bonus nor holds options.
   e. An HourlyEmployeeC has a String representation that indicates the employee's name, department, and yearly wage calculated as *hours_per_week * payrate * 52 weeks*.

Add this new type of employee to the solution to your part B, and refactor your code to maintain the *maximal reuse* property of your code. Further, you will need to modify the main method on *LabPart3_1DriverC* to create 2 hourly employees as elements 6 and 7 of the list of employees, and add this type of employee into the various loops in the main method.

*For submission, name this file LabPart3_1_DriverC.java. Further, rename all *Employee classes to *EmployeeC to avoid conflicts with previous versions. You may keep your solution in one file just as I have given you.*

2. Factory pattern, Liskov Substitution, and Inheritance by Type (15)

In part 1 you used a contrived Driver program to create employees and invoke various features. Using the **Employee* class hierarchy you created in #1 above, do the following:

A. *Create a Factory pattern interface* that can create a *Company*. Your Factory interface should:
   a. Be named *ICompanyFactory*
   b. Have a static method *public IcompanyFactory getCompanyFactory(String factoryType).* If the parameter *factoryType* is "interactive" then return an *InteractiveCompanyFactory* as specified in part B.b, and if "configurable" return a *ConfigurableCompanyFactory* as specified in B.a. If neither, return null.
   c. Have a single non-static abstract method: *public Company createCompany()*

B. *You must have two concrete implementations of IcompanyFactory::createCompany()*
   a. Name one *ConfigurableCompanyFactory*, and use a Java properties file named LabPart3_2.properties to obtain the following values:
      i. `company.hourly=<integer>`
      ii. `company.fulltime=<integer>`
      iii. `company.parttime=<integer>`
      iv. `company.name=<String>`
   b. The other should be named *InteractiveCompanyFactory*, and use a series of questions to the end user (on System.out) to gather the information necessary to create a Company. This is the exact same type of information that would be in the properties file; that is, interactively ask 4 questions to obtain the number of hourly, full time, and part time employees.
   c. Both implementations should return a new Company. Both implementations should create employees by invoking the respective class constructors using random values, specifically:
      i. Name an Employee by prefix+number, where prefix is "HE", "FTE", or "PTE" and number is just an incremental counter (you can use a private static class variable for the counter)
      ii. The department should be done by a random number generator, as should the salary or hourly wage/hours. The initial Lab3_1_DriverGiven.java should provide inspiration (lines 55-60)
   d. A Company (in Company.java) object has:
      i. A private property for the company name, and a method *getCompanyName*()
      ii. A private property (one) for the set of all Employees, as a collection of some kind (list, array, map – up to you). Provide a method *getEmployeeByNumber*(int n) that gets the nth Employee (from the counter your factory initially used) and a method *getEmployeeByName*(String name) that returns the employee with that name.
      iii. Methods *getEmployeesByType*(String type) where type is one of "HE", "FTE", or "PTE", and returns a List of Employees of that type in the company. Note that I am not giving you the full method signature – this method should return a generic, and figuring out the method signature and what you specifically return I leave up to you (there is really only one *right* way).
      iv. A method *public double computeYearlyPayroll()* that returns the sum of salary and hourly wages for all employees.
      v. A method *public double computeYearlyPayroll(String empType)* where *empType* is one of "HE", "FTE" or "PTE" and computes the expected total salary and hourly wage outlay for all employees of that type.
      vi. A method *public double computeYearlyPayroll(Departments dept)* that computes the expected total salary and hourly wage outlay for all employees in that Department.
      vii. A *public String toString()* method that lists the Company name, and the number of each type of Employee, and then the number of employees in each Department.

C. Write a driver method *public static void main(String args)* on a class LabPart3_2_Driver. This driver code should:
   a. Accept a system property (the -D flag in Java, like I did in class) named "factory", and whose value is intended to be the input parameter to the *IcompanyFactory getCompanyFactory* method specified in A.b
   b. Creates a company named "Company_ASURITE" where ASURITE is your ASURITE id, using the factory returned by step a.
   c. Outputs the result of the Company's toString method to the screen
   d. Outputs the result of calls to each of the method in B.d.iv, .v, and .vi in order. For B.d.v you need to call it 3 times. For B.d.vi. you need to call it 4 ties. Clearly prefix each call by step number and method name and param (example: "B.d.v. computeYearlyPayroll for HE employee: " + return value)
   e. Finally, using 3 calls to B.d.iii *getEmployeesByType*, loop through the return values and call toString on each employee object. Before each section put a label, like "Now printing Hourly Employees:"

D. (EXTRA CREDIT, 5 points) Modify your factory logic to make your code extensible without any modification to the existing code. Specifically, write a 3rd factory type, and provide the ability to include it, and any other future factories, into your application with no change to the existing code (Hint: I did this in class). You can get creative determining your factory's logic.

For submission, you should have files Company.java, iCompanyFactory.java , and LabPart3_2_Driver.java. Put your ICompanyFactory implementations in ICompanyFactory.java. If you do the extra credit, rename to ICompanyFactoryEC and LabPart3_2EC_Driver.java. Put all Part 3 code into one zipfile name LabPart3_ASURITE.zip. You do not need to put your code in Java packages.

Appendix A: Use Case Analysis setup

Below is a narrative describing a Course Registration system:

"As the head of information systems for Wylie College you are tasked with developing a new student registration system. The college would like a new client-server system to replace its much older system developed around mainframe technology. The new system will allow students to register for courses and view report cards from personal computers attached to the campus LAN. Professors will be able to access the system to sign up to teach courses as well as record grades.

At the beginning of each semester, students may request a course catalogue containing a list of course offerings for the semester. Information about each course, such as professor, department, and prerequisites, will be included to help students make informed decisions.

The new system will allow students to select four course offerings for the coming semester. In addition, each student will indicate two alternative choices in case the student cannot be assigned to a primary selection. Course offerings will have a maximum of ten students and a minimum of three students. A course offering with fewer than three students will be canceled. For each semester, there is a period of time that students can change their schedule. Students must be able to access the system during this time to add or drop courses. Once the registration process is completed for a student, the registration system sends information to the billing system so the student can be billed for the semester. If a course fills up during the actual registration process, the student must be notified of the change before submitting the schedule for processing.

At the end of the semester, the student will be able to access the system to view an electronic report card. Since student grades are sensitive information, the system must employ extra security measures to prevent unauthorized access.

Professors must be able to access the on-line system to indicate which courses they will be teaching. They will also need to see which students signed up for their course offerings. In addition, the professors will be able to record the grades for the students in each class."

Based on this narrative, perform the following activities:

1. Perform step 1 of the use case "How to do it" process in your notes: "Identify and describe Actors"
2. Perform steps 2 & 3: create the use case catalog and the "bubble" (use case) diagram. For each use case you should identify a title, objective, and description (feel free to cut and paste from the above text)
3. For your top 2 or 3 use cases, perform step 4: "Outline the Individual Use Cases". The "top" use cases are those you would consider to represent the essence of the system. Write the Happy Day scenario.
4. Select 1 of the use cases from the previous step and write a complete, detailed use case (step 5).
5. Revisit the above narrative – are there functional or nonfunctional requirements that do not easily fit in the use case

model? Identify these candidates for the supplementary specification.

USE CASE STEP-BY-STEP PROCESS WORKSHEET

Step 1: Identify Actors (these are highlighted in yellow on the previous page)

1. Student - one of the primary actors with goals (needs) on the system.

2. Professor – one of the primary actors with goals (needs) on the system.

3. Course Catalog – this one is the most debatable. The student can view a course catalog, which implies a separate system that manages said catalog exists, yet it is not explicitly mentioned. So if this was left off for now I'd be good with that; it most likely would show up later as you flesh the use case details out.

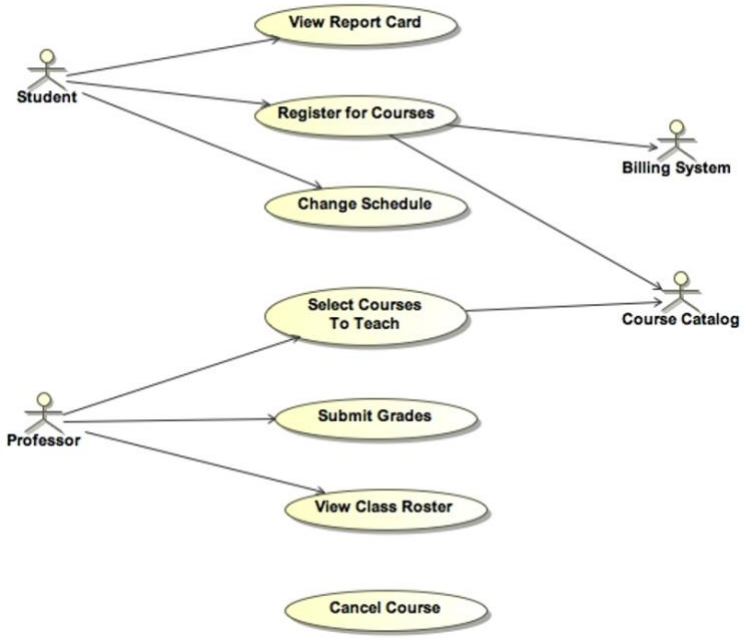
4. Billing System – used as a secondary actor.

Step 2: Identify Use Cases – Create the Use Case Catalog *(highlighted in cyan in the narrative, Primary Actors underlined)*

Use Case Name	Short Description (Goal Statement, Primary Actor, Trigger)
UC-VRC-1 View Report Card	Students need "to view an electronic report card" at the end of the semester
UC-RC-1 Register for Courses	"The new system will allow students to register for courses"
UC-SCT-1 Select Courses to Teach	"Professors must be able to…indicate which courses they will be teaching"
UC-SG-1 Submit Grades	"Professors will be able to record the grades for the students in each class."
UC-VCR-1 View Class Roster	(Professors) "will also need to see which students signed up for their class offerings"
UC-AD-1 Change Schedule	"Students must be able to access the system during this time to add or drop courses"
UC-CC-1 Cancel Course	"A course offering with fewer than three students will be canceled"

I imagine the place where you might get thrown off in this example is in the large middle paragraph – it actually does not suggest a bunch of use cases. If you thought it did, you probably fell into the trap of "functional decomposition." Really, this paragraph provides narrative detail on the registration use case for the purposes of writing a detailed UC specification (steps 4 & 5).
One subjective use case in this narrative is "Students must be able to…add or drop courses." I could see this being a distinct use case as it occurs at a distinct time from the normal registration process, and as such is difficult to capture as a variant. Whether you treat it as a separate one or as a variation, either way down the road during analysis you will have to confront the commonalities inherent in these situations – we'll get to this when we discuss factoring.
The "Cancel Course" use case is also up for some measure of debate. One could write it simply as a business rule: "The system shall cancel course offerings with fewer than three students (presumably at time XXX)." I write it as a use case as one can also see it as a scenario – an interaction that results in a goal being achieved (canceling the course offering). The tricky part is that we do not know the Primary Actor (note we do know the trigger – having less than 3 students at some given time).

Step 3: Identify Actor to Use Case relationships
Do this by drawing a use case diagram on this page. Only (and all) actors from your list of actors in Step 1 should be in the diagram, and only (and all) use cases from Step 2 should be included as well.



Step 4: Outline the Individual Use Cases (I show one Student case and one Professor case)

USE CASE: UC-RC-1 Register for Courses

Objective: PA needs to register for courses

Primary Actor (PA): Student

Trigger: PA requests a course catalog

Preconditions: Course offering has no more than 10 students registered

Post Condition(s): PA is billed for the semester registration

MAIN SUCCESS SCENARIO

1. PA requests a course catalog
2. PA selects courses for upcoming semester
3. System accepts selections and sends information to Billing System
4. System bills student for registration

USE CASE: UC-SG-1 Submit Grades

Objective: PA will be able to record the grades for students in each class

Primary Actor: Professor

Trigger: PA requests ability to record grades

Preconditions: N/A *(I always like to indicate if none apply explicitly, leaving this empty doesn't tell the reader you considered it)*

Post Condition(s): N/A *(Many folks will write a post-condition like "the students grades are recorded" but this is redundant. The completion of the Main Success Scenario or a non-failure variation means the Objective is satisfied, therefore grades are recorded).*

MAIN SUCCESS SCENARIO

1. PA accesses system
2. PA records grades

Step 5: Refine the Use Cases

Pick a Use Case from Step 4, write Alternate Scenarios & Failure Variations, plus Biz Rules and Notes.
The final use case specification would be the union of the Step 4 template for UC-RC-1 and the stuff below.

Use Case (from previous page): UC-RC-1 Register for Courses *(this is really the only one with sufficient detail in the narrative)*

Variations

Variation ID: UC-RC-Var-1: Select Alternate Courses

Preconditions: System cannot assign student to previously selected courses

Variation Scenario:

After Step 2, Main Success Scenario:	PA selects two additional alternative course choices
Modify Step 3, Main Success Scenario:	System cannot assign initial selections, System substitutes alternate choices

Note: A success variation may have different pre/postconditions but always achieves the goal of a use case. If it didn't it would be a failure variation.

Failure Variations

Variation ID: UC-RC-Var-F-1: Course is full

Preconditions: The selected course becomes full during the registration process, disabling a student's ability to register
Postconditions: PA is not registered for courses

Failure Variation Scenario:
At some time after Step 2 of the Main Success Scenario, the course reaches its maximum enrollment limit, most likely because some other Student has completed the registration process while this one is taking place (race condition).
- Alternate Step 3, 4: PA is not allowed to register, System notifies PA of failure to register, use case ends

Note that the failure scenario does not achieve the goal. If we rewrite this scenario to say the PA returns to step 1 to complete the process, then it would achieve the goal and be moved to a 2nd variation above.

Business rules

1. Course offerings will have a maximum of 10 students and a minimum of 3
2. There is a period of time (as yet unspecified) where drop/add modifications to the registration are allowed, and students must have access to the system during this time.
3. Registration is only allowed from PCs attached to the campus LAN