CENG 443 – Object-Oriented Programming Languages and Systems

Spring 2017 - Homework 1

Technical Elective Courses: The Token Dreams

Selim Temizer

Feedback : Between April 24th and April 28th, 2017
Due date : May 7th, 2017 (Submission through COW by 23:55)

In this homework, we will build an animation about CENG students dreaming that a token fairy continuously creates token packages that are vast enough for all students to take any technical electives that they want. We will use object-oriented design principles when building the animation. Basically, we will need the following classes to implement our application:

- ***Animation Entities***: These entities consist of the token fairy, technical elective courses, students and token packs. Some entities are stationary and some are mobile. Most animation entities will have various strings (like names, and other related data) displayed aside.
  - ***Token Fairy***: Represented by an image file (or more than one for flying effects). If the students have grabbed some of the available token packs, the token fairy replenishes them (randomly creates token packs and scatters them around). The total number of available token packs at any given time should roughly be a constant number.
  - ***Courses***: Stationary. Each has a different color than other courses.
  - ***Students***: Can *SEEK* available tokens around (moving to the closest token pack and grabbing it before other students), can *DEPOSIT* grabbed tokens to a random course that the student has not taken yet, can *REST* for a while occasionally, or can *LEAVE* the department if taken 2 elective courses successfully.
  - ***Token Packs***: At least 3 types of token packs should exist in the animation. The *REGULAR* token packs are colored green, and their token count is equal to the amount displayed. The *FAKE* token packs are colored blue and they are scattered around to fool rival students, with an actual count of just 1 token (independent of the amount displayed). And the third type of token packs are *BLACK* packs, which exist on black markets, and their actual token count is half of the amount displayed.
- ***Data***: Brings together animation parameters, instances of entities, and other utility fields / methods that are required for running the application.
- ***Display***: Extends *JPanel* and consists of two logical parts. At the top, technical elective courses and dynamic textual statistics about the number of students that have taken each course should be displayed. And the second part at the bottom should display the state of the animation in real time.
- ***DreamRunner***: A class that contains the main method.
- ***Vector2D***: An optional class for representing entity locations and containing various vector related utility methods.

However, we will also use the following software design patterns that will require us to extend our basic design:

- ***Factory Method / Abstract Factory***: Specific factories will create different types of token packs. Make sure that the part of the application that creates the token packs is unaware of the types of available token packs (i.e., demonstrate proper use of the factory method / abstract factory design pattern in your implementation).
- ***State***: As a minimum, 4 types of states should be prepared: *Seek*, *Deposit, Rest*, and *Leave*. At any given time during the animation, each student will be in a state picked from this list, and act accordingly. It should be possible to easily extend the system with additional states.
- ***Decorator***: Each student will be decorated with colored rectangles to denote which courses s/he has already taken (at most 2 courses can be taken by a student). Make sure that the code fragment that decorates the students is outside the *Student* class implementation, for good object oriented design.

Using the above design patterns, we should be able to:

- Prepare an environment with some predefined number of various properly initialized entities. As the animation proceeds, students will be capable of changing their states as described above. When grading, we need to clearly see the changing states during the flow of the application.
- Decorate students at run time: As students take elective courses, rectangle shaped badges should be added when drawing the students. Here, for correct usage of the decorator design pattern, make sure that the *Student* class is not aware of the fact that it will be decorated (once or more) somewhere else in the application source code.
- As a suggestion, ideally a *stepAllEntities* method in the *Data* class might ask each entity to act for one step, then might check which entities interact with each other, and decorates the students that need to be decorated. This *stepAllEntities* method might be called over and over (with a screen refresh and sleeping for a few milliseconds in between) to create the animation.

A screenshot of a sample implementation is provided in the following part of this document as a reference. We are also required to professionally **design** and briefly **document** our application. For this purpose, we will need to use a specific UML tool (StarUML) to first design the class diagram for our application, generate stub code from it, and then fill out the methods and missing implementation details. (You need to submit the UML document from StarUML application together with your source code). A detailed (but not complete) class diagram is given in the following pages to serve as a starting point and a guideline for you.

What to submit? (Use ***only ASCII characters*** when naming all of your files and folders)

1. A single ".uml" file from StarUML v1, or an ".mdj" file from StarUML v2. Name it as "***Design.uml***" (or "***Design.mdj***"). Start with an EMPTY project. Just have a single class diagram (and maybe optionally a statechart diagram and/or an activity diagram for bonuses). Don't first write the code and then reverse engineer it to get the UML. This homework aims to teach you how to go from a carefully designed UML to the code (this is what you will be hired to do for the rest of your software design careers).
2. Any other documentation that you would like to add (in a directory named "***Docs***").
3. Your source code (all in a single directory named "***Source***"). Do NOT use packages. Do NOT submit binary code or IDE created project files. Just submit your ".java" files. We should be able to compile and run your code simply by typing the following:

```
C:\...\Source>  javac  *.java
C:\...\Source>  java  DreamRunner  <any documented parameters that you might have>
```

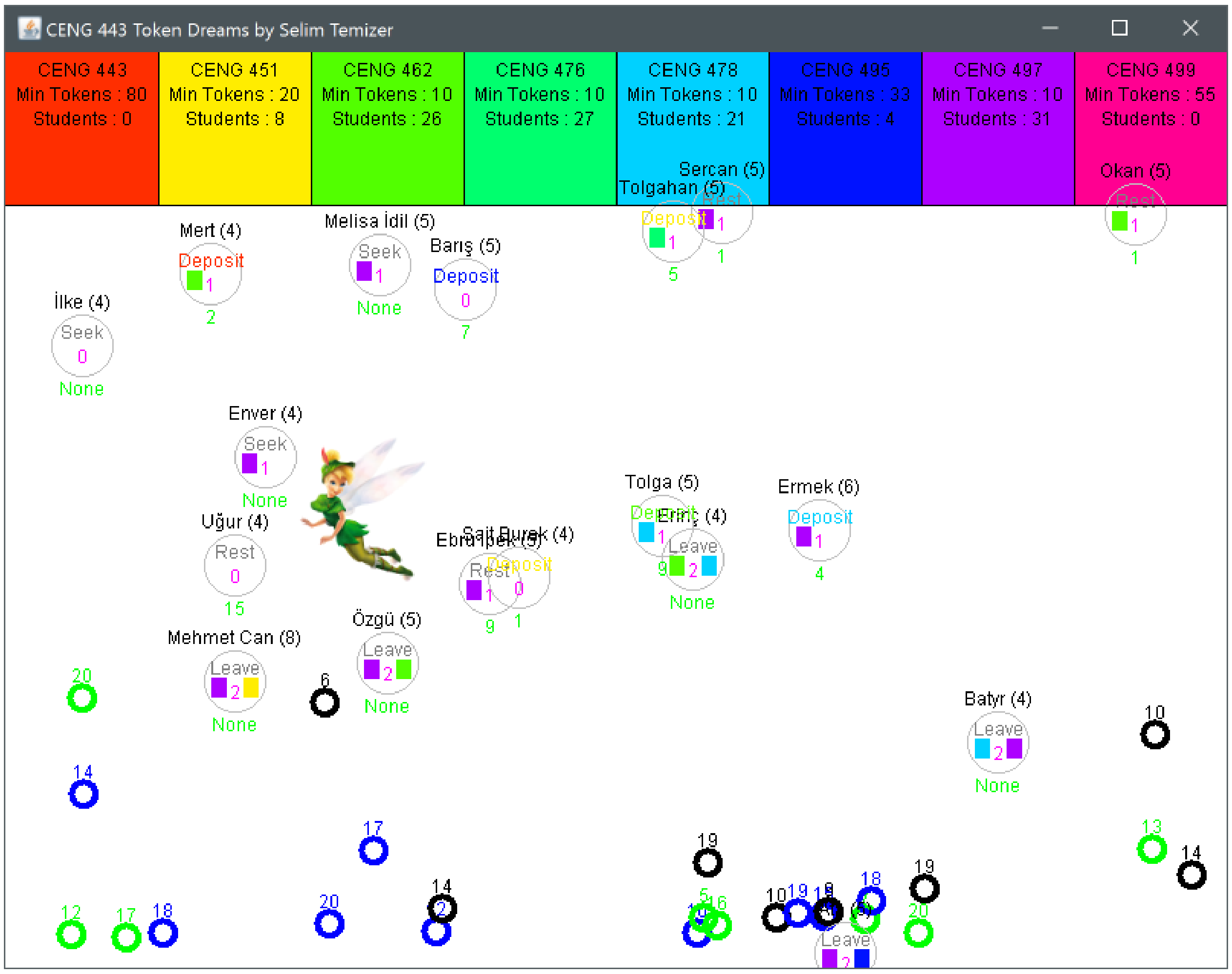
Zip the 3 items above together, give the name <ID>_<FullName>.zip to your zip file (tar also works, but I prefer Windows zip format if possible), and submit it through COW. For example:

e1234567_SelimTemizer.zip

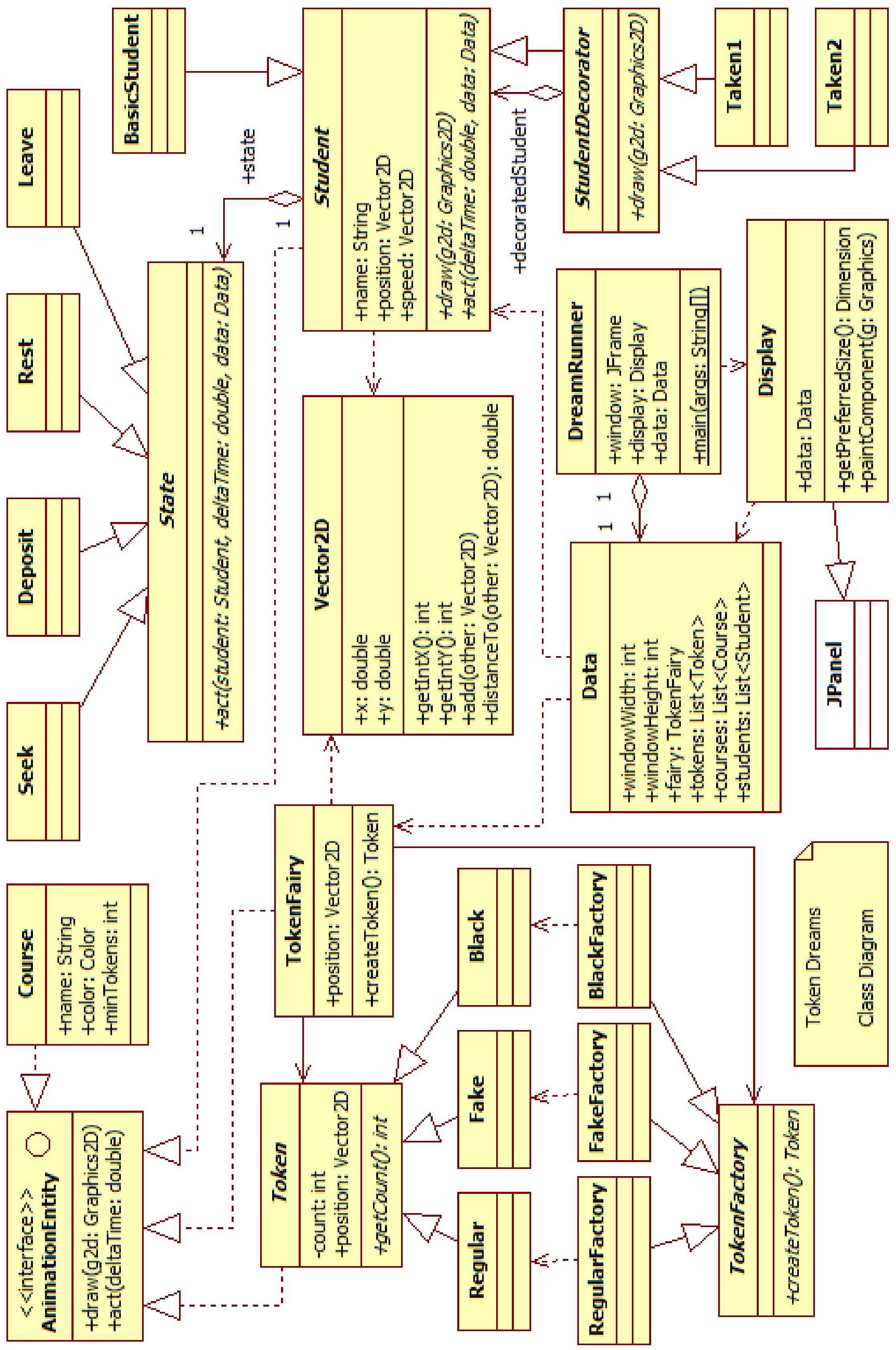
There are a number of design decisions (example: read all parameter values from editable configuration files) and opportunities for visual improvements (example: draw much nicer figures) and creative extensions (example: additional states) that are deliberately left open-ended in this homework specification. We have enough time until the deadline to discuss your suggestions and make further clarifications as necessary. There will be bonuses awarded for all types of extra effort. Late submissions will NOT be accepted, therefore, try to have at least a working baseline system submitted on COW by the deadline. Good luck.

IMPORTANT: Late submissions (even for 1 minute) will not be accepted!

We will only grade submissions on COW, and system closes automatically at due time!


CENG 443 Token Dreams by Selim Temizer
CENG 443 Min Tokens : 80 Students : 0
CENG 451 Min Tokens : 20 Students : 8
CENG 462 Min Tokens : 10 Students : 26
CENG 476 Min Tokens : 10 Students : 27
CENG 478 Min Tokens : 10 Students : 21
CENG 495 Min Tokens : 33 Students : 4
CENG 497 Min Tokens : 10 Students : 31
CENG 499 Min Tokens : 55 Students : 0
Sercan (5)
Tolgahan (5)
Okan (5)
Mert (4)
Melisa İdil (5)
Barış (5)
İlke (4)
Enver (4)
Uğur (4)
Tolga (5)
Ermek (6)
Sait Burak (4)
Ebru İpek (5)
Emir (4)
Özgü (5)
Mehmet Can (8)
Batyr (4)
Seek
Deposit
Rest
Leave
None

Sample UML Class Diagram (Draft)



Leave
Rest
Deposit
Seek
BasicStudent
State
+act(student: Student, deltaTime: double, data: Data)
+state
1
1
Student
+name: String
+position: Vector2D
+speed: Vector2D
+draw(g2d: Graphics2D)
+act(deltaTime: double, data: Data)
+decoratedStudent
StudentDecorator
+draw(g2d: Graphics2D)
Taken1
Taken2
Vector2D
+x: double
+y: double
+getIntX(): int
+getIntY(): int
+add(other: Vector2D)
+distanceTo(other: Vector2D): double
DreamRunner
+window: JFrame
+display: Display
+data: Data
+main(args: String[])
Display
+data: Data
+getPreferredSize(): Dimension
+paintComponent(g: Graphics)
1
1
Data
+windowWidth: int
+windowHeight: int
+fairy: TokenFairy
+tokens: List<Token>
+courses: List<Course>
+students: List<Student>
JPanel
Course
+name: String
+color: Color
+minTokens: int
TokenFairy
+position: Vector2D
+createToken(): Token
Black
Fake
Regular
BlackFactory
FakeFactory
RegularFactory
Token Dreams
Class Diagram
<<interface>>
AnimationEntity
+draw(g2d: Graphics2D)
+act(deltaTime: double)
Token
-count: int
+position: Vector2D
+getCount(): int
TokenFactory
+createToken(): Token