**CS 220 – Data Abstraction**

**PEX 2 – Random Sentence Generator**

**Prelim 1 Due: 2200, Lesson 15, Tuesday, 14 February**

**PEX1 Due: 2200, Lesson 18, Wednesday, 22 February**

|  |
| --- |
| Help Policy: **AUTHORIZED RESOURCES:** Any, except another cadet’s program.  **NOTE:**   * Never copy another person’s work and submit it as your own. * Do not jointly create a program unless explicitly allowed. * You must document all help received from sources other than your instructor. * **DFCS will recommend a course grade of F for any cadet who egregiously violates this Help Policy or contributes to a violation by others.**  Documentation Policy:  * You must document all help received from any source other than your instructor. * The documentation statement must explicitly describe WHAT assistance was provided, WHERE on the assignment the assistance was provided, and WHO provided the assistance. * If no help was received on this assignment, the documentation statement must state “NONE.” * If you checked answers with anyone, you must document with whom on which problems. You must document whether or not you made any changes, and if you did make changes you must document the problems you changed and the reasons why.  Vague documentation statements must be corrected before the assignment will be graded, and will result in a 5% deduction on the assignment.Turn-in Policies:  * On-time turn-in is at 2200 on the due date, same day for both M and T day sections. * Late turn-ins will receive a 25% penalty per 24 hours late unless prior arrangements have been made with your instructor. * There is no early turn-in bonus or extra credit for this assignment. |

1. Objectives

* Be able to use dynamically allocated objects
* Be able to demonstrate recursion
* Be able to create and use an ADT
* Be able to design and implement a Graphical User Interface
* Be able to design and use a Test Plan
* Be able to develop a program using pair programming
* Be able to document code using Javadoc

1. Pair Programming:

For this exercise, you must work with another cadet in the class and use the [Pair Programming](http://en.wikipedia.org/wiki/Pair_programming) approach to develop the solution for both submissions. Both students must work together ***on all parts*** of the solution and will receive the same grade on the project. You must inform your instructor of your partner selection by **Lesson 13.**

1. Background

In the past decade or so, computers have revolutionized student life. In addition to providing no end of entertainment and distractions (e.g. Facebook), computers have also facilitated all sorts of student work from English papers to calculus. One important area of student labor that has been painfully neglected is the task of filling up space in papers, Ph.D. dissertations, extension requests, etc. with important sounding and somewhat grammatically correct random sequences. An area which, until now, has been sadly neglected.

Your assignment is to construct a Java program that fills this need. The “Random Sentence Generator” is a handy and marvelous piece of technology capable of creating random sentences from a pattern known as a grammar. A grammar is a template that describes the various combinations of words that can be used to form valid sentences. There are profoundly useful grammars available to generate extension requests, generic Star Trek plots, your average James Bond movie, "Dear John" letters, and more. You can even create your own grammar. Fun for the whole family! Let’s show you the value of this practical and wonderful tool:

Tactic #1: Wear down the Instructor’s patience.

*I need an extension because I used up all my paper and then my dorm burned down and then I didn't know I was in this class and then I lost my mind and then my karma wasn't good last week and on top of that my dog ate my notes and as if that wasn't enough I had to finish my doctoral thesis this week and then I had to do laundry and on top of that my karma wasn't good last week and on top of that I just didn't feel like working and then I skied into a tree and then I got stuck in a blizzard at Tahoe and on top of that I had to make up a lot of documentation for the Navy in a big hurry and as if that wasn't enough I thought I already graduated and as if that wasn't enough I lost my mind and in addition I spent all weekend hung-over and then I had to go to the Winter Olympics this week and on top of that all my pencils broke.*

Tactic #2: Plead innocence.

*I need an extension because I forgot it would require work and then I didn’t know I was in this class.*

Tactic #3: Honesty.

*I need an extension because I just didn't feel like working.*

**What is a grammar?**

A grammar is a set of rules for some language, be it English, the Java programming language, or an invented language. If you go on to study computer science, you will learn much more about languages and grammars in a formal sense. For now, we will introduce a particular kind of grammar called a Context Free Grammar (CFG). Here is an example of a simple grammar:

The Poem grammar

{

**<start>**

The **<object> <verb>** tonight. ;

}

{

**<object>**

waves ;

big yellow flowers ;

slugs ;

}

{

**<verb>**

sigh **<adverb>** ;

portend like **<object>** ;

die **<adverb>** ;

}

{

**<adverb>**

warily ;

grumpily ;

<**adverb**> and painfully ;

}

According to this grammar, two possible poems are "The big yellow flowers sigh warily tonight" and "The slugs portend like waves tonight." Essentially, the strings in brackets (<>) are variables which expand according to the rules in the grammar.

More precisely, each string in brackets is known as a "non-terminal". A non-terminal is a placeholder that will expand to another sequence of words when generating a poem. In contrast, a "terminal" is a normal word that is not changed to anything else when expanding the grammar. The name “terminal” indicates it is a dead-end— no further expansion is possible from here.

A definition consists of a non-terminal and its set of "productions" or "expansions" each of which is terminated by a semi-colon ';'. There will always be at least one and potentially several productions that are expansions for the non-terminal. A production is just a sequence of words, some of which may be non-terminals. A production can be empty (i.e. just consist of the terminating semi-colon) which makes it possible for a non-terminal to expand to nothing. The entire definition is enclosed in curly braces '{' '}'. The following definition of "<verb>" has three productions:

{

**<verb>**

sigh **<adverb>** ;

portend like **<object>** ;

die **<adverb>** ;

}

Comments and other irrelevant text may be outside the curly braces and should be ignored (quickly advancing over the extraneous stuff outside the braces is a good use of the Scanner's skipping functions, I might add). All the components of the input file: braces, words, and semi-colons will be separated from each other by some sort of white space (spaces, tabs, newlines), so you will be able to use those as delimiters when parsing the grammar. And you can discard the white-space delimiter tokens since they are not important. In order to read the grammar files, you will find the Scanner routines quite handy.

Once you have read in the grammar, you will be able to produce random expansions from it. You begin with the single non-terminal <start>. For a non-terminal, consider its definition, which contains a set of productions. Choose one of the productions at random. Take the words from the chosen production in sequence, (recursively) expanding any which are themselves non-terminals as you go. For example:

<start>

The <object> <verb> tonight. --expand <start>

The big yellow flowers <verb> tonight. --expand <object>

The big yellow flowers sigh <adverb> tonight. --expand <verb>

The big yellow flowers sigh warily tonight. --expand <adverb>

Since we are choosing productions at random, doing the derivation a second time might produce a different result and running the entire program again should also result in different patterns.

1. Preliminary Exercise

* Design a way to store a grammar in a data structure that could be used to solve the random sentence generation problem. In order to store the grammar, you can use a data structure from the Java Collections Framework (e.g. ArrayList, LinkedList, etc.). An effective solution will have multiply nested Lists. It may have a List of Lists, or even a List of List of Lists… You are allowed to use the Java Collections Lists here because you will need to use the template or generic capabilities of the list. For example, you may need something like the following somewhere:

ArrayList<ArrayList<String>> rules;

* Next, design a linked list to hold the Strings in your expanded sentence. This linked list must be implemented entirely by you. You cannot use the ArrayList or LinkedList from the Java collections framework for this portion. Your linked list only needs to implement the functionality required to expand and store the grammar. You do not need to implement the entire List interface from the Java Collections Framework.
* Document your entire design (Grammar ADT, Java Collections used, your linked list, and the resulting expanded random sentence) with a UML diagram.
* Using your design from above, create the java methods required to solve the problem. You do not have to implement any methods for the preliminary turn-in. You just need the method declarations, with complete and detailed Javadoc comments, of course.
* Create JUnit test procedures to fully test your linked list implementation. You should test all the methods in your linked list. The JUnit test methods should be fully implemented for the preliminary turn in, but the methods they will be testing do not need to have any functionality implemented.
* Design a user interface for your application. This video will help you learn how: <http://www.youtube.com/watch?v=LFr06ZKIpSM&feature=channel_page>. You do not need to implement any of the GUI functionality for the preliminary turn in.

1. Preliminary Submission Requirements

* Fill in your name and the documentation statement on the prelim grade sheet file at the end of this document and save it as a separate file in the top level of your NetBeans project folder!
* Only one submission is required per pair programming team.
* Your NetBeans project name must be Lastname1-Lastname2PEX2.
* Zip this entire folder to Lastname1-Lastname2PEX2.zip and submit it using the Webpost link on the left side of the course web page.
* Submit only one file for the programming exercise, a zip file containing your entire NetBeans project with the UML class diagram and the PRELIM Grade Sheet document in the top-level folder. Your Java project file should include a fully implemented Graphical User Interface (GUI). The buttons don’t have to be functional at this point.

1. Programming Exercise

Implement the full functionality of the RSG based on your design from the preliminary turn in and feedback from your instructor. Your application must allow the user to load a grammar file (several have been provided for you) and repeatedly generate random sentences from that grammar. They should also be able to load another grammar file or select another grammar from a set of pre-loaded grammars. A simple plan of attack could be the following:

* Implement and fully test your linked list class using your JUnit tests and feedback you received from the preliminary turn in. Expanding the sentence correctly will depend upon your linked list working correctly.
* Implement the functionality to select a grammar file, read the contents, and load the data into your grammar data structure.
* Implement the functionality to expand your sentence starting with the non-terminal <start> into a random sentence. **You must do this recursively using your linked list.**

Remember, this is a pair programming assignment. **Both students must be present during the development** with one as the driver and one as the navigator. You must track how much time is spent by each student as the driver and as the navigator (see pair programming log). You should switch roles every 15-30 minutes.

1. Programming Exercise Submission Requirements

* Fill in your name and the documentation statement on the PEX grade sheet file at the end of this document and save it as a separate file in the top level of your NetBeans project folder!
* Only one submission is required per pair programming team. You must include your pair programming log in your submitted turn in.
* Your NetBeans project name must be Lastname1-Lastname2PEX2.
* Zip this entire folder to Lastname1-Lastname2PEX2.zip and submit it using the Webpost link on the left side of the course web page.
* Submit only one file for the programming exercise, a zip file containing your entire NetBeans project with the PEX Grade Sheet document and pair programming log in the top-level folder.

1. Helpful hints

Successfully solving this problem will require a well-designed ADT to support your implementation. Work with your partner on your design and discuss it with your instructor. Think of the things in the problem statement that might translate into classes/objects. You have a grammar. The grammar has rules. Rules consist of a label (i.e. non-terminal symbol) and a list of terminals and non-terminals.

**CS 220 – PEX 2 Prelim – Grade Sheet Name:** Delphia / Johnson

Assessments

Criteria Self Instructor Points

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| UML diagram describes a reasonable class hierarchy for this problem | | **10** |  | **10** |
| UML diagram contains all required information in the correct format | | **5** |  | **5** |
| GUI is laid out in Netbeans and provides all required elements to solve the problem | | **3** |  | **3** |
| JUnit tests are present to fully test the student’s linked list implementation | | **7** |  | **7** |
| Skeleton Java code exists with full Javadoc for all classes specified in the UML design | | **5** |  | **5** |
| **Subtotal:** | | **30** |  | **30** |
| **Adjustments** | **Vague/Missing Documentation:** | **−0** | **−** | **− 2** |
| **Submission Requirements Not Followed:** | **−0** | **−** | **− 2** |
| **Late Penalties:** | **−0** | **−** | **25/50/75%** |
| **Total w/adjustments:** | **30** |  |  |

Documentation Statement:

Cadet Matthew Johnson and Cadet Gavin Delphia completed this assignment while holding each other’s hands the entire time. We did not receive help from anyone else.

Comments from Instructor:

**CS 220 – PEX 2 – Grade Sheet Name:**

Assessments

Criteria Self Instructor Points

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Linked List JUnit tests fully test linked list functionality | |  |  | **6** |
| Linked List provides required methods and passes all JUnit tests | |  |  | **12** |
| Load and parse grammar file successfully | |  |  | **10** |
| Grammar data structure fully supports loading of all grammar elements | |  |  | **15** |
| Recursive generation of random sentence | |  |  | **12** |
| GUI is properly implemented and support application functionality | |  |  | **5** |
| Pair programming log provided that demonstrates roughly equal drive time by each student | |  |  | **10** |
| All methods have adequate javadoc comments | |  |  | **5** |
| In line comments present and accurately describe what the code does | |  |  | **5** |
| **Subtotal:** | |  |  | **80** |
| **Adjustments** | **All Java code meets specified standards:** | **−** | **−** | **− 8** |
| **Vague/Missing Documentation:** | **−** | **−** | **− 4** |
| **Submission Requirements Not Followed:** | **−** | **−** | **− 4** |
| **Late Penalties:** | **−** | **−** | **25/50/75%** |
| **Total w/adjustments:** |  |  |  |

Documentation Statement:

Comments from Instructor:

Pair Programming Log

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Driver** | **Navigator** | **Start Time** | **Stop Time** | **Duration (mins)** |
| LtCol Boleng | Dr. Bower | 2210 | 2230 | 20 |
| Dr. Bower | LtCol Boleng | 2230 | 2300 | 30 |
| LtCol Boleng | Dr. Bower | 2300 | 2325 | 25 |
| Dr. Bower | LtCol Boleng | 2325 | 2340 | 15 |
|  |  |  |  |  |
|  |  |  |  |  |

Programming Time:

LtCol Boleng: 45 mins

Dr. Bower: 45 mins

Total-------------------------------------------

90 mins