CSE231 Class Project – Operational Concept

Ref: "Big O" Notation; YouTube video, Derek Banas;
https://www.youtube.com/watch?v=V6mKVRU1evU&index=9&list=PLGLfVvz_LVvReUrWr94U-ZMgjYTQ538nT

Objectives

The Class Project has several objectives: (1) to understand how to take an Operational Concept for a System (including a set of specific questions), create a UML Model of the System; (2) implement the UML Model in Java code using Test-Driven Development (TDD); (3) learn how to develop UML Models and translate the UML Model into Java code by Independent Teams using TDD; (4) learn how to integrate this Team Java code into a working application;

Overview

The Class Project consists of 3 Parts.

Part I – Comparison & Ranking of Data Structures by Time Complexity ("Big O" Notation)

This section requires the development of Java code to compare various Java Collections Framework (JCF) Data Structures ("containers"). These Data Structures will have increasing numbers of elements. You will compare them with respect to the Time Complexity for the execution of specific algorithms. You will also determine the actual time of execution of each algorithm. You will then rank each Data Structure using the "Big O" notation discussed in class and contained in JCF_Monograph_03A and the "Big O" reference document posted separately on Moodle. Recall that the "Big O" notation shows the dependency on a specific parameter that influences the way a particular algorithm or data structure scales as the number of elements increases.

Part II – Choosing the Correct Algorithm to Implement a System Requirement

This section demonstrates the Project Team's ability to analyze a specific System requirement and apply the correct algorithm with the appropriate Data Structure to implement System requirement. The System requirements will take the form of questions posed to the Project Team.

Part III – JUnit Testing in the Development of JCF Data Structures

This section involves (1) constructing a JUnit test class; (2) conducting a JUnit test to assert some functional behavior of a specific JCF algorithm and underlying JCF Data Structure. The Project Team must show how the JUnit test class facilitates the selection and construction of the appropriate JCF algorithm and underlying JCF Data Structure.

Class Project Architecture

The System for the Class Project is a System to register students for a class at OU. The System consists of 3 Subsystems: (1) Display; (2) Middleware; and (3) Database. These Subsystems implement the well-known UML Design Pattern, "Model-View-Controller (M-V-C)". This design pattern provides complete decoupling of the three Subsystems – which means the internal construction of one Subsystem does not affect another Subsystem. This remains true as long as the "operational contract" between specific Subsystems is maintained. This "operational contract" simply means that the interfaces between Subsystems must be completely agreed between Subsystem Teams.

Here is a System-Level "White Box" sequence diagram for a UML use case named "Create_Data_Structure_UC". The Astah UML model is posted on Moodle along with this Operational Concept.

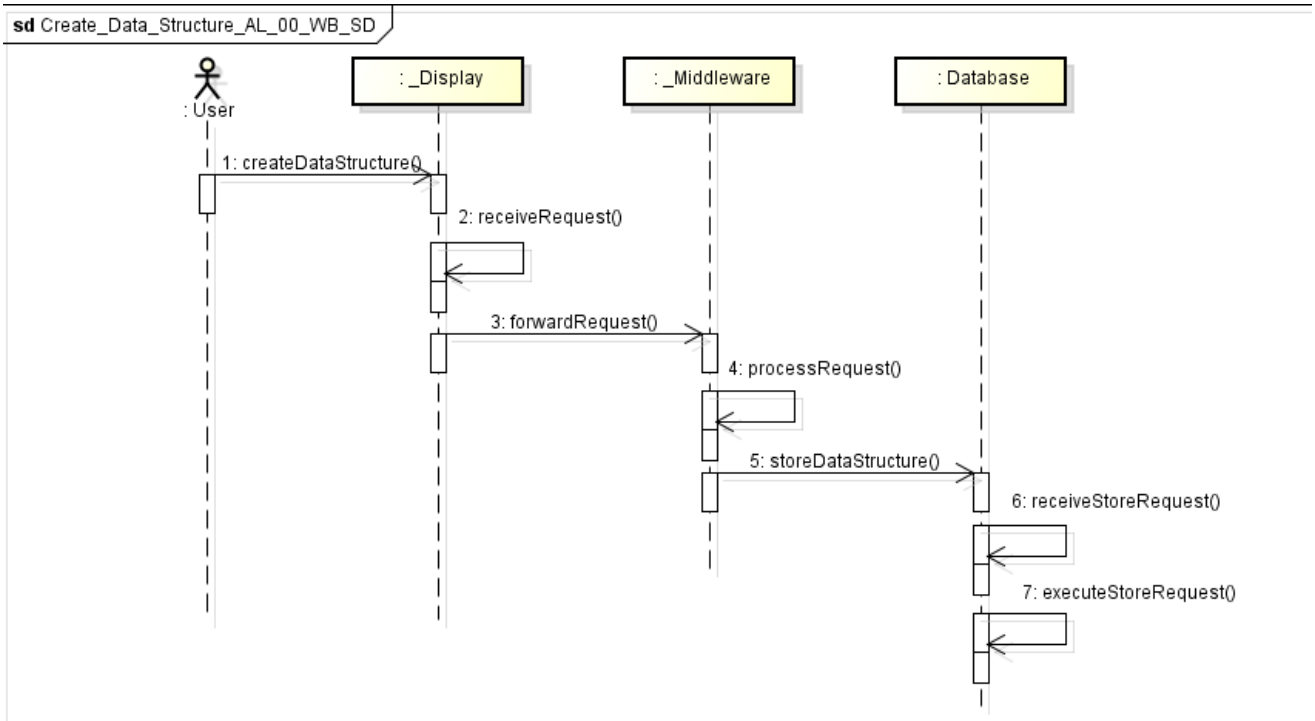

Figure 1. the AL_00_WB_SD on the use case "Create_Data_Structure_UC";

Subsystem Roles

Display

Display acts as an interface for the User to the System. The User will enter commands & data from the command line and receive printed responses on the command line. The Display Team should consider using the "java.util.Scanner" class to capture all User keyboard inputs. In summary, the Display Subsystem

(1) forwards the commands & data the User enters from the command line;

(2) receives responses to commands and data inputs from Middleware back to the command line;
(3) is responsible for formatting and printing the Middleware command & data responses to the command line;
(4) does no processing of User commands & data; all User inputs are passed to Middleware for processing;

Middleware

Middleware implements the System "business rules" (System functional requirements), including any necessary calculations. Middleware applies the System business rules to commands & data received from Display. Middleware executes the commands & data inputs from the command line by creating the ordered Data Structure and filling it with the provided data. Middleware then passes the completed Data Structure to Database for permanent storage. Middleware also responds to data retrieval requests from Display for specific data which may be located in specific Data Structures stored in Database.

In summary, Middleware

(1) processes & executes the commands & data inputs passed as requests from Display;
(2) creates and fills Data Structures as a result of processing requests from Display;
(3) initiates "store" requests to Database to permanently store Data Structures created & filled by Middleware;
(4) initiates data retrieval requests to Database in response to processing data retrieval requests from Display;
(5) does not initiate any requests to Display;

Database

Database provides storage services for permanent storage of all Data Structures created by Middleware. Database also responds to Middleware data retrieval requests for specific data.

In summary, Database

(1) provides permanent storage & retrieval services as requested by Middleware;
(2) does not initiate any requests to Middleware; it only responds Middleware requests;

Sectional Details

Part I (A) – ArrayList<E> and LinkedList<E>

Fill the following Data Structures with Random Numbers from 100 to 1000. Each Data Structure shall have (1) 100K elements; (2) 200K elements; (3) 400K elements;

1. Create an ArrayList<Integer> and a LinkedList<Integer>.
2. Conduct a selection sort and a bubble sort of each Data Structure.
3. Measure the actual elapsed time to do each sort.

4. Show by actual return data what the approximate "Big O" parameter dependency relationship is for each Data Structure for each sort. Return "times through" (the loop) for binary search below.
5. Conduct a linear and binary search of each Data Structure (after the sort).

Part I (B) – Binary Tree

1. Now construct a Binary Tree of (1) 100K Nodes; (2) 200K Nodes; (3) 400K Nodes;
2. Generate the key value pairs as follows:
    a. generate non-duplicate random numbers (ints) between 400K and 800K for the Node key;
    b. generate additional random numbers between 200K and 600K, cast as Strings;
    c. Construct the Binary Tree using these random ints (Node keys) and random Strings (Node values);
3. Select 3 random String values and conduct (1) a preorder search; and (2) an in order search for each value;
4. Record the actual elapsed time; also print the number of Nodes searched;
5. Show by actual data what the approximate "Big O" relationship is for the Binary Tree;

Part I (C) Hash Table

1. Now construct a Hash Table of (1) 100K elements; (2) 200K elements; (3) 400K elements; You can use the same set of key-values as the Binary Tree above; use these as the input to the Hash Table to construct the Hash code; print the elapsed time to construct the Hash Table;
2. Do a search for the same 3 values;
3. Record the actual elapsed time; also print the number of elements searched;
4. Show by actual data what the approximate "Big O" parameter dependency relationship is for the Hash Table;

Part II - Solving Problems Using Algorithms

1. Suppose you want to store 10 students that register for a class in the order that they registered. The class is overbooked. The students below registered in the order shown;

Registration Order	Grizzly ID	Name	Major	GPA	Thesis Grade
01	677422	Jones	IT	3.82	95
02	177993	Smith	IT	3.47	78
03	444811	Breaux	CS	3.95	98
04	113625	Brady	CS	3.77	92
05	382707	Rominske	CS	3.82	79
06	447447	Hardy	IT	3.68	99
07	661284	Kominsky	IT	3.23	70
08	855462	O'Brien	IT	3.44	85
09	223344	Chamberlain	CS	3.99	96
10	348689	Grant	CS	3.88	99

Figure 2. student registration data;

Suppose you are the Instructor who must cut the last 5 students to register. Construct a JCF Algorithm Structure to show

(1) which Grizzly IDs and associated names are cut;
(2) the complete description of the last student cut;
(3) the Grizzly IDs and names of the students still remaining in the class;

2. Now suppose you want to process the remaining students for a scholarship based on the order that they registered for class. Assume that the first 3 to apply are successful. Construct JCF Algorithm Structure to show which of the remaining students did not receive a scholarship.
3. Suppose all 10 students are reinstated into the class. They submit their individual Thesis in the same order as registration.
    a. Store these Thesis results in a Binary Tree;
    b. Traverse the Binary Tree using a Preorder Traverse and print out the name and Thesis grade of each class member to the command line;
    c. Which students stored within the Binary Tree has a GPA < 3.60 and a Thesis grade less than 90? Print out the Name and Grizzly ID of each student in this category to the command line.

Part III – JUnit Testing

Referring to Part II above:

1. Write the JUnit test class that will test that the last person to register for the class is the first person cut from the class. Confirm by executing the JUnit test. Print all results to the command line. Make sure you show the JUnit swing gui with appropriate tests run shown.
2. Write a JUnit test class that test whether the first person to register for the class wins a scholarship or not. Confirm by executing and printing results to the command line. Make sure you show the JUnit swing gui with appropriate tests run shown.