Object Oriented Programming Assignment

2018/2019

Student Name: Valerija Holomjova

Course Code: CPS2004

Table of Contents

Task 1 – C++

Overview

The following program reads and executes specific commands from a given file line by line. A binary search tree is created where each node represents an animal object which stores the data and properties of a certain animal. The program uses inheritance and polymorphism in order to create and store the animal objects. Each line in the file should either be empty or contain an Insert, Find or Remove a command to manipulate the nodes in the tree. After reading the end of the file, the name and type of the animal nodes in the tree are displayed using inorder traversal. The tree also has implementations for the preorder and postorder traversal in the 'animals.cpp' file, but these are not called in the program.

Exceptions and error messages are used to notify the user of invalid arguments. In cases of not being able to find/ or delete a node, the program will continue to run however a custom error message will be displayed in the terminal – notifying the user a command couldn't be fulfilled and the reason why.

Important things to note:

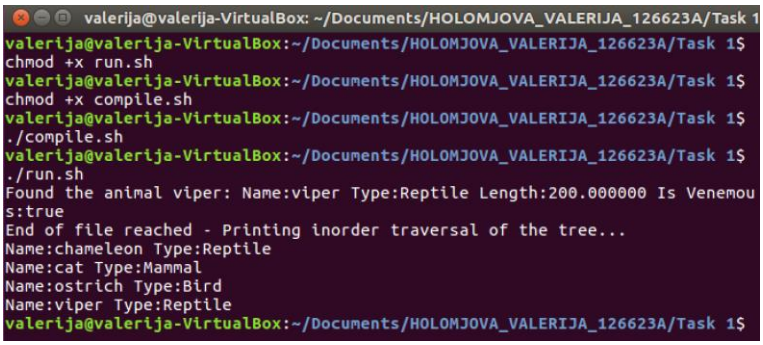
- It is important to note that all animals are stored in relation to their animal length in a binary search tree manner. If a new node has the same length as an existing node in the tree, it will not be added, and the user will be notified.
- All commands must start with an uppercase letter and be of the form "Insert", "Find" or "Remove" and have the appropriate number of arguments as specified in the assignment document (for eg. the Insert command needs 5 arguments).
- When inserting a node, all animal types must start with a lowercase letter and be of the form "mammal", "reptile" or "bird".
- The names and Boolean attributes (eg. "can-fly") are also case sensitive and should be written in lowercase. If the user inputs an animal with an uppercase name, in order to find the same animal, the letter case must be identical to the one that was used to input the animal name.

Compiling and Running the Program

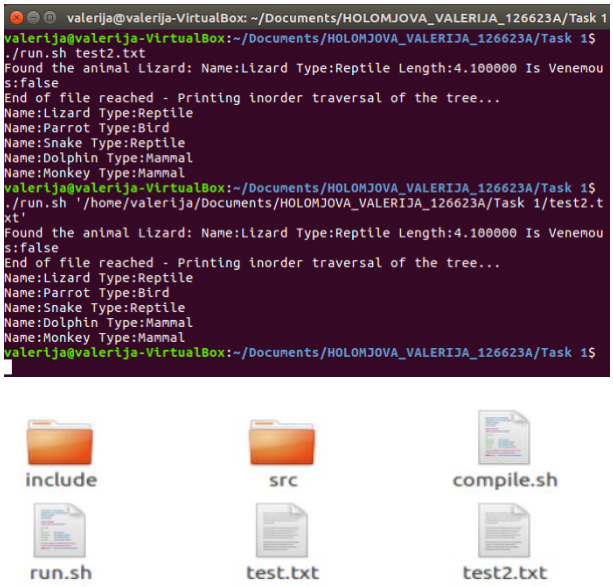
To run and compile the program navigate to the Task 1 directory where the run.sh and compile.sh files are located. First, execute the following commands to change the permissions of the files to ensure they are executable:

- chmod +x compile.sh
- chmod +x run.sh

Then compile the files using the command "./compile.sh" followed by the command "./run.sh" to execute the program. Below is an example of how the program was compiled and executed.



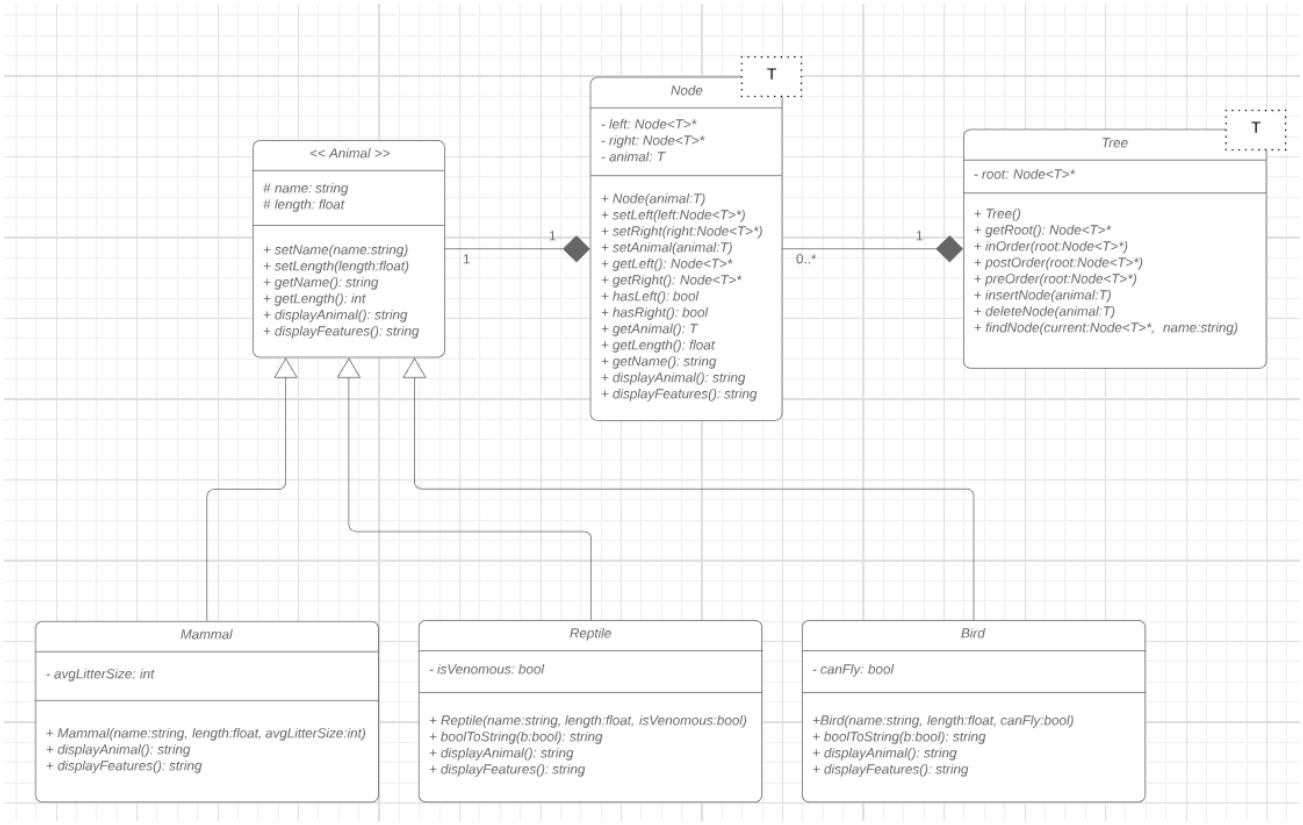
If no argument is inputted after the "./run.sh" command, the default test file will be run. If a valid file path is added after "./run.sh", the program will execute the commands in the given file path. It is important to note that the absolute path of the file must be given or the path of the file with respect to the Task 1 directory since the executable ("main.exe") is created there. For instance, in the example below I have created a file called "test2.txt" which includes valid commands and placed it in the Task 1 directory. I then executed it using its file name since it was placed in the same directory as where the executable will be made. I also executed the file using its absolute file path.



(Test2.txt being placed in the Task 1 directory)

UML Class Diagram

Below is a UML class diagram of the application.



Declaration/Header Files (.h)

In the next section, I will be briefly explaining the files inside the Include folder.

header.h

The following file contains the libraries and namespaces that will be used in the program. The contents of the file are displayed below.

```
#ifndef ANIMALS_HEADER_H
#define ANIMALS_HEADER_H

//All libraries used.
#include <iostream>
#include <string>
#include <fstream>
#include <vector>
#include <sstream>
#include <algorithm>
#include <iterator>
#include <exception>
#include <math.h>

using namespace std;

#endif
```

animals.h

The following file contains the declarations of the animal class and its subclasses – the mammal, reptile and bird class. The animal class is the base class and contains variables to store the name and length of an animal since these are required and common between the subclasses. It also contains two virtual members which will be overridden by an overriding member in the derived classes. Below is part of the file where the animal class is declared.

```
#ifndef ANIMALS_ANIMALS_H
#define ANIMALS_ANIMALS_H

#include "header.h"

//Animal - base class.
class Animal {
protected:
    string name;
    float length = 0;
public:
    void setName(string name);
    void setLength(float length);
    string getName();
    float getLength();
    virtual string displayAnimal();
    virtual string displayFeatures();
};
```

The subclasses (mammal, reptile and bird) are derived from the animal base class and contains unique variables that match the animal they are representing. For instance, the mammal class has an

integer to store the average litter size and the reptile class has a Boolean variable to store whether the object is venomous or not. Each subclass also has an overriding member which will display all the features of the subclass object – including the name and length derived from the base class. These functions were made overriding members so that different features could be added to the string with respect to the type of animal. The string from the feature displaying functions are used for Inorder traversal of the tree or to display the features of a found node. Below is part of the file where the animal subclasses are declared.

```
//Subclasses of animal - mammal, reptile and bird which are derived from the animal
base class.
class Mammal: public Animal{
    int avgLitterSize;
public:
    Mammal(string name, float length, int avgLitterSize);
    //Overridden functions to display animal features.
    string displayAnimal() override;
    string displayFeatures() override;
};

class Reptile: public Animal{
    bool isVenomous;
public:
    Reptile(string name, float length, bool isVenemous);
    string boolToString( bool b );
    //Overridden functions to display animal features.
    string displayAnimal() override;
    string displayFeatures() override;
};

class Bird: public Animal{
    bool canFly;
public:
    Bird(string name, float length, bool canFly);
    string boolToString(bool b);
    //Overridden functions to display animal features.
    string displayAnimal() override;
    string displayFeatures() override;
};

#endif
```

tree.h

The following file contains the declarations of the node and tree class templates. The node class template has two variables that accept pointers to another node of the same type which represent the left and right child of the node. It also has a generic data type variable which will be used to store a polymorphic object, specifically a pointer to an animal base class which is assigned to the addresses of an animal subclass object such as a mammal. That way each node will have access to the members of the subclass objects but are still regarded as the same type. The node includes getter/setter functions, functions to check whether the left or right child of the node is empty, and functions to retrieve information from the members of the animal object such as its name or its subclass such as its features. Below is part of the file where the node class and its members are declared.

```
template <class T> class Node{
    Node<T> *left;
    Node<T> *right;
    T animal;
public:
    explicit Node(T animal);
    void setLeft(Node<T> *left);
    void setRight(Node<T> *right);
    void setAnimal(T animal);
    Node<T>* getLeft();
    Node<T>* getRight();
    bool hasLeft();
    bool hasRight();
    T getAnimal();
    //Getting from Animal Class Functions
    float getLength();
    string getName();
    string displayAnimal();
    string displayFeatures();
};
```

The tree class template is used to store the node objects that are described above. It uses a node pointer to keep track of the root of the tree, then it traverses through each node using pointers to the left and right children nodes, which are stored in each node object. The template contains member functions that resemble a binary search tree such as outputting the tree using inorder, postorder or preorder traversal or inserting, finding and deleting a node. Below is part of the file where the tree class template and its members are declared.

```
//Tree generic class template.
template <class T> class Tree{
    Node<T>* root;
public:
    Tree();
    Node<T>* getRoot();
    //------------- Printing the Tree -------------------
    void inOrder(Node<T> * root);
    void postOrder(Node<T> *root);
    void preOrder(Node<T> *root);
    //------------- Inserting/Deleting/Finding the Tree -------------------
    void insertNode(T animal);
    Node<T>* findNode(Node<T> *current, string name);
    void deleteNode(T animal);
};

#endif
```

parser.h

The following file includes all animal and tree object definitions to be used in the ‘parse.cpp’ file. It has declarations of the functions that will be needed to parse and execute a file. It also contains declarations to functions that create new animal subclass objects and return a pointer to them so that their memory can be referenced by an animal base class pointer and added to the binary search tree. The contents of the file are displayed below.

```
#ifndef ANIMALS_PARSER_H
#define ANIMALS_PARSER_H

#include "header.h"
//Including animal and tree object definitions.
#include "animals.h"
#include "tree.h"
#include "../src/tree.cpp"

void parsefile(string fileName);
Mammal* getMammal(vector<string> tokens);
Reptile* getReptile(vector<string> tokens);
Bird* getBird(vector<string> tokens);
vector<string> getTokens(string line);

#endif
```

Implementation Files (.cpp)

main.cpp

This file contains the main function of the program and calls the 'parsefile()' function which is derived from the 'parser.h' file and implemented in the 'parser.cpp' file. The main function should always receive an argument since the test file path or user input is always passed by the 'run.sh' script, however, in case the 'run.sh' script isn't used, the function will run the test file if an argument wasn't given, or else consider the file path to be the user's additional argument. Below is the implementation of the main function.

```
#include "../include/parser.h"

int main(int argc, char* argv[]) {
    //Executes the default test file or the given argument.
    if(argc == 1){
        cout << "No argument was provided...\n";
        parsefile("test.txt");
    } else {
        parsefile(argv[1]);
    }
    return 0;
}
```

animals.cpp

The following file includes the implementation of the animal class and its subclasses. The code starts by including the "animals.h" file which has all the declarations of the members of the animal base class and its subclasses. Next in the code, the getter/setter functions of the animal base class are implemented. Then it defines two virtual members of the animal base class which return an empty string since they will be overridden by the derived classes. A constructor was not created for the animal base class since in the "parser.cpp" file, only subclasses were instantiated and immediately assigned to animal base class pointers. Below is part of the file where the methods of the animal base class were implemented.

```
#include "../include/animals.h"

//Animal - base class.
void Animal::setName(string name){
    this->name = move(name);
}
void Animal::setLength(float length){
    this->length = length;
}
string Animal::getName(){
    return name;
}
float Animal::getLength(){
    return length;
}
//Virtual members - can be redefined in a derived class.
string Animal::displayAnimal(){
    return " ";
}
string Animal::displayFeatures(){
    return " ";
}
```

The code proceeds by implementing the methods of the subclasses. Each subclass constructor accepts a name, length and attribute specific to the type of animal. Then all three subclasses have two overridden functions to display their features. The 'displayAnimal()' function returns a string with the animal name and it's type, whereas the 'displayFeatures()' returns all the information of the animal including its name, length and specific attribute. The reptile and bird function have an extra function called 'boolToString' that returns the string "true" or "false" depending if the variable given is true or false. This was implemented so that the outputting the features could be displayed in a neater way instead of numbers. Below is an implementation of one of the subclasses.

```
Reptile::Reptile(string name, float length, bool isVenemous) {
    this->length = length;
    this->name = move(name);
    this->isVenomous = isVenemous;
}
//Convert a boolean into true or false.
string Reptile::boolToString(bool b) { return b ? "true" : "false";}
//Redefinition of base class function - overridden functions.
string Reptile::displayAnimal() {return "Name:"+name+" Type:Reptile";}
string Reptile::displayFeatures() {return "Name:"+name+" Type:Reptile Length:"+to_string(length)+" Is
Venemous:"+boolToString(isVenomous)+"";}
```

tree.cpp

The following file includes the implementation of the tree and node class templates. The code starts by including the "tree.h" file which has all the declarations of the members of the tree and node class templates. Next in the code, the getter/setter functions of the node template class are implemented followed by functions that calls the methods of the animal object that is stored in every node object in order obtain its name, length or retrieve strings displaying its features. Subsequently, two functions (hasLeft() and hasRight()) are implemented to return true or false based on whether a node object has a left child or a right child. Below are the implementation methods of the node class.

```
//------------------- Node Functions -------------------
template <class T>
Node<T>::Node(T animal) {
    this->animal = animal;
    left = right = nullptr;
}
template <class T>
void Node<T>::setLeft(Node<T> *left) {
    this->left = left;
}
template <class T>
void Node<T>::setRight(Node<T> *right) {
    this->right = right;
}
template <class T>
void Node<T>::setAnimal(T animal) {
    this->animal = animal;
}
template <class T>
Node<T>* Node<T>::getLeft() {
    return left;
}
template <class T>
Node<T>* Node<T>::getRight() {
    return right;
}
template <class T>
T Node<T>::getAnimal() {
    return animal;
}
template <class T>
```

```
float Node<T>::getLength() {
    return animal->getLength();
}
template <class T>
string Node<T>::getName() {
    return animal->getName();
}
template <class T>
string Node<T>::displayAnimal() {
    return animal->displayAnimal();
}
template <class T>
string Node<T>::displayFeatures() {
    return animal->displayFeatures();
}
template <class T>
bool Node<T>::hasLeft() {
    if(!left)
        return false;
    return true;
}
template <class T>
bool Node<T>::hasRight() {
    if(!right)
        return false;
    return true;
}
```

inOrder()/preOrder()/postOrder() function

Next the methods of the tree template class are defined. When constructing a new tree object, the root of the tree is set to null, which is also an indication that the tree is empty. Three functions are implemented to print the tree in inorder, preorder or postorder traversal. The functions recursively loop through each node depending on the traversal method. For instance, for the 'inOrder()' function, the function prints the left subtree recursively, then prints the animal features of the current node, then recursively prints the right subtree. If the current node is null, nothing is displayed, which acts as the base case for the recursive function.

This is the same for the other two functions, however the order of how the nodes are printed are different, for instance, In the 'preOrder()' function, the animal features of the current node are printed first, then the function prints the left and right tree recursively. Below is a part of the implementation of the tree template class.

```
//------------------- Tree Functions -------------------

template <class T>
Tree<T>::Tree(){
    root = nullptr;
}
template <class T>
Node<T>* Tree<T>::getRoot(){
    return root;
}

//------------------- Printing Nodes -------------------
//Recursively print the left subtree, then the root, then the right subtree of each node.
template <class T>
void Tree<T>::inOrder(Node<T> * root){
    if (root != NULL) {
    inOrder(root->getLeft());
    cout << root->displayAnimal() << '\n';
    inOrder(root->getRight());
    }
}
```

```
//Recursively print the root, then the left subtree, then the right subtree of each node.
template <class T>
void Tree<T>::preOrder(Node<T> *root){
    if (root != nullptr) {
        cout << root->displayAnimal() << '\n';
        preOrder(root->getLeft());
        preOrder(root->getRight());
    }
}

//Recursively print the left subtree, then the right subtree, then the root of each node.
template <class T>
void Tree<T>:: postOrder(Node<T> *root){
    if (root != nullptr) {
        postOrder(root->getLeft());
        postOrder(root->getRight());
        cout << root->displayAnimal() << '\n';
    }
}
```

insertNode() function

In the insert function of the tree, a generic data type object is accepted, which will be a pointer to an animal object. At the start, a new node containing the animal object is created to and pointed to. If the root of the tree is null indicating that the tree is empty, the root of the tree is set to the new node. Otherwise, the function will iterate through the tree and try find a suitable position for the new node based on the animal object's length. It does this by accessing the left or right child of each node depending on whether the length of the new nodes animal object is less than or greater than the current nodes animals' length respectively. If the selected child of that node is empty, the current nodes selected child is set to the new node, otherwise it sets the current node to the selected child and loops again.

For instance, if the new nodes animal length is smaller than the current nodes animal length, it checks whether the current nodes left child is empty. If it is empty, it sets the current nodes left child to the new node otherwise, it sets the current node to the left child and compares the new nodes left with it again. Note that if the new nodes animal length is the same as one of the nodes, the function will output an error saying such item already exists. Hence, each animal node should be of different length. Below is the definition of the 'insertNode()' function of the tree template class.

```
//------------------- Inserting Nodes -------------------
//Inserts an new node into a tree given a node.
template <class T>
void Tree<T>:: insertNode(T animal){
    //Creates a pointer to a new animal node.
    Node<T> *node = new Node<T>(animal);
    //If the tree is empty, set the root of the tree to the new node.
    if(root == nullptr){
        root = node;
        return;
    }
    //Iterates through each node in the tree to find a suitable position for the new node.
    Node<T> *current = root;
    while(current != nullptr){
        //Comparing the new animals length with the current nodes length.
        if(animal->getLength() < current->getLength()){
            //If the current node has no left children, set the new node as the left child of the
current node.
            if(!current->hasLeft()){
                current->setLeft(node);
                break;
            }
            //Otherwise, keep searching for a free position in the left children of current node.
            current = current->getLeft();
```

```
        } else if (animal->getLength() > current->getLength()){
            //If the current node has no right children, set the new node as the right child of the
current node.
            if(!current->hasRight()){
                current->setRight(node);
                break;
            }
            //Otherwise, keep searching for a free position in the right children of current node.
            current = current->getRight();
        } else {
            //If the new animal has the same length as current node, item won't be added.
            cerr << "Item already exists.\n";
            delete node;
            break;
        }
    }
}
```

findNode() function

The find function of tree tries to find a node with an animal name that matches a string argument that's passed. Since the nodes of the tree are stored by length, the function tries to check all the nodes of tree until it finds it. It does so recursively by checking whether the current node matches the given string. If the current node matches the given string, it returns the current node. Otherwise it calls the function recursively for the left and right subtree of the current node until the function reaches a null pointer (the base case), indicating the end of a branch. If no nodes were found recursively in the left or right subtree, null pointer is returned. Below is the definition of the 'findNode()' function of the tree template class.

```
//------------------- Finding Nodes -------------------
//Finds a node with the same name and returns a pointer to it.
template <class T>
Node<T>* Tree<T>:: findNode(Node<T> *current, string name) {
    //Base case, return null if current node is empty.
    if(current == nullptr){ return nullptr; }
    //If the current node has the same name, return the node.
    if(current->getName() == name){
        return current;
    } else {
        Node<T> * found;
        //Searches for the node in the left and right children of the current node.
        if((found = findNode(current->getLeft(),name)) != nullptr){
            return found;
        } else if ((found = findNode(current->getRight(),name)) != nullptr){
            return found;
        } else {
            //If node wasn't found in the left or right children, null is returned.
            return nullptr;
        }
    }
}
```

deleteNode() function

The delete function of the tree deletes a node from the tree given an animal object. If the tree is empty, the function displays an error and returns. The function starts by trying to find the node to be deleted in the tree based on the given animal objects length and the current nodes animal length. If the given animal objects length is smaller than the current nodes, it sets the current node to the left subtree and if the given animal objects length is larger than the current nodes animal length, it sets the current node to the right subtree. During this process it keeps track of the current nodes

parent node and stops when the current nodes animal length matches the given animals' length or when an empty node is reached. If an empty node is reached the function will return. Otherwise, the function will proceed to delete the node in one of three methods depending on the number of children it has. The code below illustrates part of the 'deleteNode()' function where the current node is set to the node to be deleted and its parent is kept track of.

```
//Deletes a node from the tree given a node.
template <class T>
void Tree<T>:: deleteNode(T animal) {
    //If the root is null, function returns since tree must be empty.
    if(root == nullptr){cerr << "Tree is empty - can't delete node.\n"; return;}
    //Enters a while loop to set the current node as the node to be deleted and keep track of it's
parent.
    Node<T> *current = root;
    Node<T> *parent = nullptr;
    while(current){
        //Searches in the left or right children of current node based on the length of the node to be
deleted.
        if(animal->getLength() > current->getLength()){
            parent = current;
            current = current->getRight();
        } else if (animal->getLength() < current->getLength()) {
            parent = current;
            current = current->getLeft();
        } else {
            //Breaks when current node matches the node to be deleted.
            break;
        }
    }
    //If the current node is null, node couldn't be found so functions returns.
    if(!current){ return;}
```

Next the function checks if the node to be deleted has no children. If this condition is met and the node is the root of the tree (by checking if the parent is a null pointer), then the root is set to a null pointer and the function returns. Otherwise, if it has no children but is the left child of the parent node, then the left child of the parent is set to a null pointer and the function returns. If it has no children but is the right child of the parent node, then the right child of the parent is set to a null pointer and the function returns. Note that the data of the current node is destroyed in all three cases after the node is 'removed' from the tree. The code below illustrates part of the 'deleteNode()' function where the node to be deleted has no children.

```
//If the node has no children.
    if(!current->hasLeft() && !current->hasRight()){
        //If it is the root, set the root to null.
        if(parent == nullptr){root = nullptr; delete current; return;}
        //If it is the left node of the parent node, set left node of parent to null.
        if(parent->getLeft() == current){
            parent->setLeft(nullptr);
        //If it is the right node of the parent node, set right node of parent to null.
        } else {
            parent->setRight(nullptr);
        }
        delete current;
        return;
    }
```

Next the function checks whether the node has only child. If the node only has a right child and it is the root of the tree (by checking if the parent is a null pointer), then the root of the tree is set to the right child of the current node. Otherwise the left or right node of the parent node is set to the right child of the current node, depending on whether the current node is the left or right node of the parent node respectively. The data of the current node is destroyed after its right child takes its place in the tree. The same process happens if the node only has a left child, except instead of the right child of the current node taking its place, the left node of the current node takes its place. The

code below illustrates part of the ‘deleteNode()’ function where the node to be deleted has one child.

```
//If the node only has right children.
    if(!current->hasLeft()){
        //If it is the root, set the root to the right children.
        if(parent == nullptr){root = current->getRight(); delete current; return;}
        //If it is the left node of the parent node, set left node of parent to the deleted node's
right children.
        if(parent->getLeft() == current){
            parent->setLeft(current->getRight());
        //If it is the right node of the parent node, set right node of parent to the deleted node's
right children.
        } else {
            parent->setRight(current->getRight());
        }
        delete current;
        return;
    }

    //If the node only has left children.
    if(!current->hasRight()){
        //If it is the root, set the root to the left children.
        if(parent == nullptr){root = current->getLeft(); delete current; return;}
        //If it is the left node of the parent node, set left node of parent to the deleted node's
left children.
        if(parent->getLeft() == current){
            parent->setLeft(current->getLeft());
        //If it is the right node of the parent node, set right node of parent to the deleted node's
left children.
        } else {
            parent->setRight(current->getLeft());
        }
        delete current;
        return;
     }
```

If neither of the conditions above were met, the function assumes the node has two children. If this the case, the function proceeds to find the inorder successor of the node to be deleted, by finding smallest value node in the right branch of the node to be deleted. It does this by accessing the right child of the node to be deleted and looping through each left child node until the next left child is empty. It keeps track of the parent of the inorder successor so that after the inorder successor is found, it’s parent sets its left child to a null pointer. Next, the animal data from the node to be deleted is replaced with the animal data of the inorder successor so that all the children of the node to be deleted remain in their appropriate place and then the memory of the temporary inorder successor is removed. The code below illustrates part of the ‘deleteNode()’ function where the node to be deleted has two children.

```
     //If the node has left and right children - get the inorder successor.
    Node<T> *nextParent = current->getRight();
    Node<T> *next = current->getRight();
    //Gets the smallest value (next) in the right branch of the node to be deleted and its parent
(nextParent).
    while(next->hasLeft()){
        nextParent = next;
        next = next->getLeft();
    }
    //Removes the successor node from it's parent.
    nextParent->setLeft(nullptr);
    //Replacing the data of node to be deleted with the successor node.
    current->setAnimal(next->getAnimal());
    delete next;
}
```

parser.cpp

parsefile() function

This file is responsible for parsing and executing a file derived from a given file path. The function tries opening a file stream for reading or returns the function with an error message if the file cannot be found. Next a tree object is declared of animal base class pointers which will be used to store all the inserted animal nodes. Th function proceeds to loop through each line of the file by using the standard library function 'getline()' to read each line from the file. For validation purposes, the '/r' is removed from the end of the line since empty lines from the 'getline()' function are given a '/r' character instead of an empty string. The function proceeds to remove any empty lines, so that they will not be considered as commands in the next stage and causing the program to crash. Next, the 'parsefile()' function proceeds by obtaining the first argument in the line assuming each argument must be separated using a space character (' ') and checks whether the user wants to insert, remove or find a node using the equality operator. Below is part of the implementation of the parsefile() function where the file is opened and a while loop is entered to read each line and retrieve the first argument of each line.

```
#include "../include/parser.h"

void parsefile(string fileName){
    //Opening a file for reading using filename and returning if it can't be open.
    ifstream file(fileName);
    if(!file) { cerr << "Cannot open input file.\n"; return; }

    //Declaring an object of a the tree class that will store pointers to the animal class.
    Tree<Animal *> tree;

    //Reading and executing the file line by line.
    string line;
    while (getline(file, line)) {
        //Remove /r from end of line.
        if(line[line.size() - 1] == '\r'){
            line.resize(line.size()-1);
        }
                //If the current line is empty, continue reading the next line.
        if(line.empty()){ continue;}
        //Get the first argument in the line.
        string token = line.substr(0,line.find(' '));
```

If the user wants to insert a node, the code will call the 'getTokens()' function which splits a given line into a vector of string tokens and returns them. Each string in the tokens vector is considered as an argument in the command, so the size of the vector is validated to ensure the number of arguments inputted match the number of arguments the Insert command needs to have. If the vector does not have the correct number of arguments, an exception is thrown. Otherwise, the code proceeds by determining the type of animal to be added by comparing the second argument to a "mammal", "reptile" or "bird".

It then calls a particular function specific to the type which will return a pointer to a newly created subclass object that uses the argument and matches the type. For instance, if the second argument matches the string "mammal", the 'getMammal()' function is called and accepts the tokens as arguments in order to create a new mammal object and then return it. Once the subclass object is made, an animal base class pointer assigns itself to the new subclass object. This method of polymorphism is used so that the tree could store the animal object (since all nodes must be of the same type) and the node can maintain its subclass properties. All animal types are inserted into the tree the same way, the difference is that each animal subclass gets its own function to be created

and returned and the animal base class pointer points to the appropriate subclass object in each case. It is important to note that if the first argument in the line did not match “Insert”, “Remove” or “Find”, the function will throw an invalid argument exception stating that the given commands were unrecognizable. Below is part of the implementation of the ‘parsefile()’ function where the animal nodes are inserted into the tree.

```
//Check whether the first argument is Insert, Find or Remove.
        if(token == "Insert"){
            //Tokenizes the string and checks there's a correct number of arguments.
            vector<string> tokens = getTokens(line);
            if(tokens.size() != 5){ throw invalid_argument("Invalid number of arguments.");}
            //Checks the type of animal to be inserted into the tree.
            if(tokens[1] == "mammal"){
                Animal *animal = getMammal(tokens);
                tree.insertNode(animal);
                continue;
            } else if (tokens[1] == "reptile"){
                Animal *animal = getReptile(tokens);
                tree.insertNode(animal);
                continue;
            } else if (tokens[1] == "bird"){
                Animal *animal = getBird(tokens);
                tree.insertNode(animal);
                continue;
            } else {
                throw invalid_argument("Can't recoginze command. " + tokens[1]);
            }
```

Next the function checks whether the first argument of the line indicates the user wants to find a node in the tree. If this is true, the code tokenizes the current line into a vector of tokens representing each argument of the line and checks whether the size of the vector matches the number of arguments necessary for the “Find” command. If the number of arguments doesn’t match, an exception of invalid argument is thrown stating there was an invalid number of arguments. If they do match, the tree calls it’s member function ‘findNode()’ and uses it’s root as the starting point (since the function is recursive) and the second argument as the name of the node to be found. If a null pointer is returned from the function, the node with the given name could not be found and an error is displayed to the user, but the program does not terminate. Otherwise, the member function ‘displayFeatures()’ of the found node is used to display the nodes animal object features such as its name, length, type and unique attribute. Below is part of the implementation of the ‘parsefile()’ function where the program tries to find a node in the tree.

```
} else if (token == "Find"){
            //Tokenizes the string and checks there's a correct number of arguments.
            vector<string> tokens = getTokens(line);
            if(tokens.size() != 2){ throw invalid_argument("Invalid number of arguments.");}
            //Checks if the animal exists in the tree.
            if(tree.findNode(tree.getRoot(),tokens[1]) == nullptr){
                cerr << "Could not find the animal " + tokens[1] + "\n";
            } else {
                //Displays information about the animal.
                cout << "Found the animal " + tokens[1] + ": ";
                cout << tree.findNode(tree.getRoot(),tokens[1])->displayFeatures() + "\n";
            }
```

Next the function checks whether the first argument of the line indicates the use wants a node to be deleted in the tree. First, the code checks whether the node exists using the ‘findNode()’ function similarly as described above, except in this case if the node is found, it calls the delete node function. If the node isn’t found using the ‘findNode()’ function an error is displayed notifying the user the node couldn’t be deleted and the program continues. Below is part of the implementation of the ‘parsefile()’ function where the program tries to delete a node.

```
} else if (token == "Remove"){
            //Tokenizes the string and checks there's a correct number of arguments.
            vector<string> tokens = getTokens(line);
            if(tokens.size() != 2){ throw invalid_argument("Invalid number of arguments.");}
            //Checks if the animal exists in the tree, if it does, deletes the node using the found
node.
            if(tree.findNode(tree.getRoot(),tokens[1]) == nullptr){
                cerr << "Could not find and delete the animal " + tokens[1] + "\n";
            } else {
                tree.deleteNode(tree.findNode(tree.getRoot(),tokens[1])->getAnimal());
            }
        } else {
            throw invalid_argument("Command not recognized. " + token);
        }
    }
```

At the end of the 'parsefile()' function, after each line of the file has been read and executed, the function calls the inorder member function of the tree depending on whether the tree is empty or not. If the tree is empty and has no nodes, it will output an error saying the tree is empty and there are no nodes to display the inorder traversal of the tree. Otherwise, it calls the member function using the root of the tree as the node starting point since the inorder function is recursive. Below is the final part of the implementation of the 'parsefile()' function where the program displays the inorder traversal of the tree.

```
//If no nodes were inputted, notifies the user that the tree is empty, otherwise prints the inorder
traversal of the tree.
    if(tree.getRoot() == nullptr){
        cerr << "Tree is empty - No nodes to display inorder traversal of the tree.\n";
    } else {
        cout << "End of file reached - Printing inorder traversal of the tree...\n";
        tree.inOrder(tree.getRoot());
    }
}
```

getMammal() / getBird()/ getReptile() functions

Each of these functions create a distinct animal subclass object given a set of arguments. The 'getMammal()' creates a mammal object by attempting to convert the length (4th argument) and average litter size (5th argument) into float and integer variable respectively. Each of these functions try to convert the length argument into a float variable, but only mammals have an average litter size attribute which is stored in the object as an integer. If the arguments cannot be converted to a float and integer, an invalid argument exception is thrown, and an error is displayed in the console. Otherwise, a new mammal object is created using the 3rd, 4th and 5th argument as a valid string name, float length and integer average litter size variables. The new mammal object is then returned.

The 'getReptile()' function is the same as the mammal function except for instead of checking the 5th argument as a valid integer, it tries to convert the 5th argument into a Boolean by checking whether it is the string "venomous" or "non-venomous" and setting a variable called 'isVenemous' to true or false respectively. If it is neither of the two strings it throws an invalid argument exception. Otherwise a new reptile object is created and returned using the 3rd, 4th and 5th argument as a valid string name, float length and Boolean 'isVenomous' variables.

The 'getBird()' function is identical to the reptile function but instead it creates a Boolean by checking whether the 5th argument is the string "can-fly" or "cannot-fly". The new bird object is then returned. Below is the implementations of the 'getMammal()', 'getReptile()' and 'getBird()' functions.

```
//Given a string of argument tokens, returns an object of Mammal class.
Mammal* getMammal(vector<string> tokens){
    float length = 0;
    int avgLitterSize = 0;
    //Tries to convert the animal length and litter size into a float and integer variable
respectively.
    try {
        length = stof(tokens[3]);
        avgLitterSize = stoi(tokens[4]);
    } catch (const invalid_argument &ia) { cerr << "Invalid argument: " << ia.what() << '\n'; }
    //Create a new object of Mammal class using arguments.
    Mammal *mammal = new Mammal(tokens[2], length, avgLitterSize);
    return mammal;
}

//Given a string of argument tokens, returns an object of Reptile class.
Reptile* getReptile(vector<string> tokens){
    float length = 0;
    bool isVenomous;
    //Tries to convert the animal length into a float variable.
    try{
        length = stof(tokens[3]);
    } catch (const invalid_argument &ia) { cerr << "Invalid argument: " << ia.what() << '\n'; }
    //Tries to identify whether the animal is venomous or non-venomous.
    if(tokens[4] == "venomous"){
        isVenomous = true;
    } else if (tokens[4] == "non-venomous"){
        isVenomous = false;
    } else { throw invalid_argument("Can't read command :" + tokens[4]); }
    //Create a new object of Reptile class using arguments.
    Reptile *reptile = new Reptile(tokens[2], length, isVenomous);
    return reptile;
}

//Given a string of argument tokens, returns an object of Bird class.
Bird* getBird(vector<string> tokens){
    float length = 0;
    bool canFly;
    //Tries to convert the animal length into a float variable.
    try{
        length = stof(tokens[3]);
    } catch (const invalid_argument &ia) { cerr << "Invalid argument: " << ia.what() << '\n'; }
    //Tries to identify whether the animal can fly or not.
    if(tokens[4] == "can-fly"){
        canFly = true;
    } else if (tokens[4] == "cannot-fly"){
        canFly = false;
    } else { throw invalid_argument("Can't read command :"  + tokens[4]); }
    //Create a new object of Bird class using arguments.
    Bird *bird = new Bird(tokens[2], length, canFly);
    return bird;
}
```

getTokens() function

This function splits a given string into a vector of strings using the space character (‘ ’) as a delimiter. A vector was used to its dynamic storage abilities. The vector of strings is then returned. Below is the implementation of the ‘getTokens()’ function.

```
//Splitting a string into tokens by ' ' character.
vector<string> getTokens(string line){
    vector<string> tokens;
    stringstream ss(line);
    string token;
    while(getline(ss,token,' ')){
            tokens.push_back(token);
    }
    return tokens;
}
```

Bash Scripts (.sh)

compile.sh

This file compiles all the .cpp files which are found in the src folder and creates an executable ("main.exe") to be used in the "run.sh" bash script. Below is a screenshot of the contents of the file.

```
#!/bin/bash

#Compiling all cpp files in the src folder
g++ -Wall -std=c++11 -o main src/*.cpp
```

run.sh

This file executes the program by running the executable "main.exe" and specifying the file path to be used by checking whether an additional argument was passed when running the bash script. If the user adds an additional argument, the script executes the program using the added argument as the file path. If no arguments were provided, the path of the test file is used when running the program. Below is a screenshot of the contents of the file.

```
#!/bin/bash

#If additional argument wasn't given, run the default test file
if [ "$#" -eq "0" ]
then
    ./main "test.txt"
else
    ./main "$1"
fi
```

Task 2 – Java

Overview

The following program reads and executes specific commands from a given file line by line. Each restaurant that is added is stored as an object in an array list due to its flexible storage. Objects are created to store the data of each order list. Another object is used to store all the order lists encountered. The prices of items in an order is obtained from the array list of restaurant objects. Also, each order list object is implemented in a linked list fashion, where each order object is pointing to another order object.

The program uses an observer design pattern to keep track of the total money spent and highest profited restaurant in a single order list as well as all the order lists combined. When an order is added to an order list, the order list observer is updated. When an order list is added to the object containing all the order lists, the observer of all order lists is updated. Note that when an order list is added, the total money spent in the order list is displayed. After the entire file is read, the highest profited restaurant and the total money spent in all the order lists is displayed. Exceptions are used to notify the user of invalid arguments.

Important things to note:

- It is important to note that all arguments are case sensitive and should contain the appropriate number of arguments. The valid commands are “BeginRestaurant”, “Item”, “EndRestaurant”, “BeginOrderList”, “BeginOrder”, “OrderItem”, “EndOrder”, “EndOrderList”.
- It is important to note that a restaurant name in the BeginOrder command must match the an existing restaurant with that naame and be of the same letter case. A restaurant which an order needs must be defined before an order.
- The delivery/takeaway/both argument must be written in lower case for both the “BeginRestaurant” and “BeginOrder” command (eg. “takeaway”/ “delivery”/ “both”).

Compiling and Running the Program

To run and compile the program navigate to the Task 2 directory where the run.sh and compile.sh files are located. First, execute the following commands to change the permissions of the files to ensure they are executable:

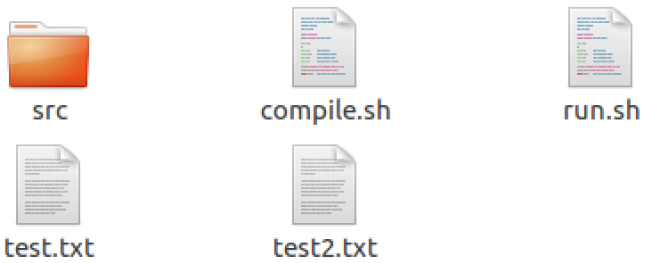
- chmod +x compile.sh
- chmod +x run.sh

Then compile the files using the command "./compile.sh" followed by the command "./run.sh" to execute the program. Below is an example of how the program was compiled and executed.

```
valerija@valerija-VirtualBox:~/Documents/HOLOMJOVA_VALERIJA_126623A/Task 2$
chmod +x run.sh
valerija@valerija-VirtualBox:~/Documents/HOLOMJOVA_VALERIJA_126623A/Task 2$
chmod +x compile.sh
valerija@valerija-VirtualBox:~/Documents/HOLOMJOVA_VALERIJA_126623A/Task 2$
./compile.sh
valerija@valerija-VirtualBox:~/Documents/HOLOMJOVA_VALERIJA_126623A/Task 2$
./run.sh
Added a new order list with a total of: 21.0€.
--------------------------------------------------------------------------------
The highest revenue restaurant is 'Cikku' with a total revenue of 13.0€.
The total price all order lists is 21.0€.
valerija@valerija-VirtualBox:~/Documents/HOLOMJOVA_VALERIJA_126623A/Task 2$
```

If no argument is inputted after the "./run.sh" command, the default test file will be run. If a valid file path is added after "./run.sh", the program will execute the commands in the given file path. It is important to note that the absolute path of the file must be given or the path of the file with respect to the directory in which the "runner.java" file is (in the src folder) since it contains the main function and its bytecode will be created there. For instance, in the example below I have created a file called "test2.txt" containing some valid commands and placed it in the Task 2 directory. I then executed the file using its absolute file path and then by using the path of the file with respect to where the "runner.java" file is located (in the src folder).

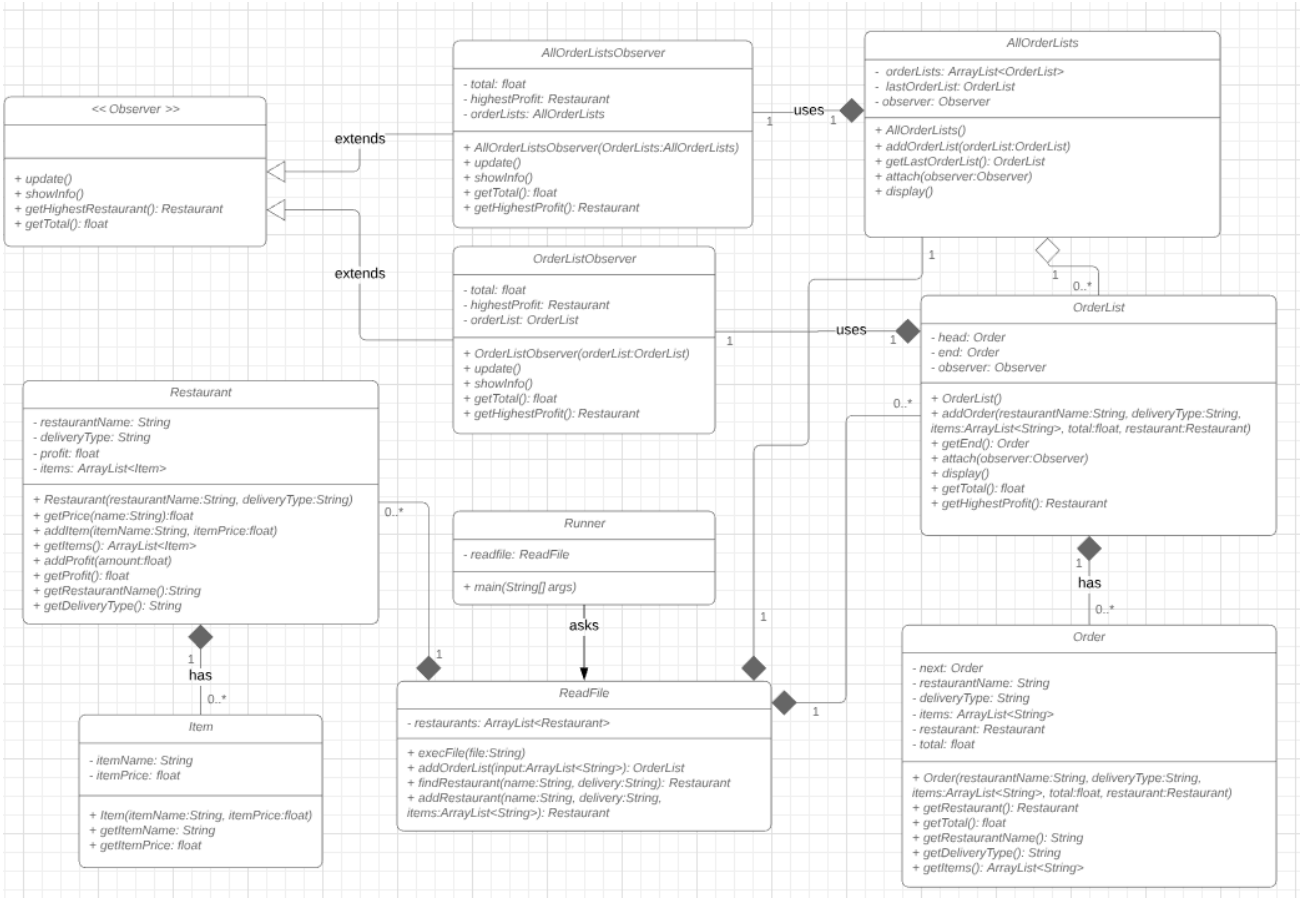
```
valerija@valerija-VirtualBox:~/Documents/HOLOMJOVA_VALERIJA_126623A/Task 2$
./run.sh ../test2.txt
Added a new order list with a total of: 18.8€.
Added a new order list with a total of: 17.0€.
--------------------------------------------------------------------------------
The highest revenue restaurant is 'Agliolio' with a total revenue of 29.5€.
The total price all order lists is 35.8€.
valerija@valerija-VirtualBox:~/Documents/HOLOMJOVA_VALERIJA_126623A/Task 2$
./run.sh '/home/valerija/Documents/HOLOMJOVA_VALERIJA_126623A/Task 2/test2.t
xt'
Added a new order list with a total of: 18.8€.
Added a new order list with a total of: 17.0€.
--------------------------------------------------------------------------------
The highest revenue restaurant is 'Agliolio' with a total revenue of 29.5€.
The total price all order lists is 35.8€.
valerija@valerija-VirtualBox:~/Documents/HOLOMJOVA_VALERIJA_126623A/Task 2$
```



(Test2.txt being placed in the Task 2 directory)

UML Class Diagram

Below is a UML class diagram of the application.



Source Files (.java)

The following section of the documentation will proceed by describing each (.java) file in the 'src' folder.

Runner.java

This file contains the main function of the program and calls the 'execFile()' function which is derived from a ReadFile object which is implemented in the 'ReadFile.java' file. The main function should always receive an argument since the test file path or user input is always passed by the 'run.sh' script, however, in case the 'run.sh' script isn't used, the function will run and execute the test file if an argument wasn't given, or else consider the file path to be the user's additional argument. Below is the implementation of the main function. It also throws an exception if any error occurs in the 'ReadFile()' function. Below is the code of the 'Runner.java' file

```
public class Runner {
    private static ReadFile readFile = new ReadFile();

    public static void main(String[] args) {
        //Checks whether to execute a given command line argument or the default test file.
        try{
            if(args.length == 0){
                readFile.execFile("../test.txt");
            } else {
                readFile.execFile(args[0]);
            }
        } catch (Exception e){
            e.printStackTrace();
        }
    }
}
```

ReadFile.java

The following file defines a class which contains methods responsible for reading and executing a given file path. An array list is defined at the start of the class and will hold a list of restaurant objects. These restaurant objects hold the information of each restaurant such as the restaurant name, delivery type and items which are read from the file.

execFile() function

The 'execFile()' is the function which will accept the file path and execute each line. It starts by creating a AllOrderLists object to store each order list from the given file. It also creates an observer for the AllOrderLists object to keep track of the highest profited restaurant and overall money spent in all the order lists.

The code proceeds to try to create a buffer from the given file path and read each line in a while loop. If there is a problem opening the file an exception is thrown. Each line in the file is stripped from '\n' or '/r' characters and trimmed for whitespaces. If a line is empty, the loop reiterates and the next line in the file is read. Next, the line is tokenized by the space ' ' character into an array of string tokens. Below is part of the implementation of the execFile() function where the file is opened and a while loop is entered to read and parse each line.

```
class ReadFile {
    //Stores all the restaurants from the file.
    private ArrayList<Restaurant> restaurants = new ArrayList<>();

    //This function reads from a given file and parses and executes each line.
    void execFile(String file){
        //Stores all the order lists and attaches an observer.
        AllOrderLists allOrderLists = new AllOrderLists();
        new AllOrderListsObserver(allOrderLists);

        //Creates a new buffer to read each line of the file.
        try (BufferedReader br = new BufferedReader(new FileReader(file))) {
            while (br.ready()) {
                String line = br.readLine();

                //Validation to remove empty lines and trim possible whitespaces.
                line = line.replace("\n", "").replace("\r", "").trim();
                if(line.length() == 0){ continue; }

                //Tokenizes the string to check for commands.
                String[] tokens = line.split("\\s+");
```

Next the function checks whether the first argument of the line is to create a new restaurant or add a new order list and has the appropriate number of arguments. If the arguments do not follow these expectations, an exception is thrown stating that the commands given in the line could not be recognized.

If the first argument of the line matches “BeginOrderList” and the number of arguments is valid, the function extracts the lines between the “BegindOrderList” and “EndOrderList” into an array list of strings where each string is a line. If the “EndOrderList” is not found, an exception is thrown. The ‘addOrderList()’ function is used to take the array list of line and return an order list object which is then added to the AllOrderLists object. The function the re-iterates and the next line in the file is read. Below is part of the implementation of the execFile() function where a new order list is created.

```
//Creating a new order list.
if(tokens[0].equals("BeginOrderList")  && tokens.length == 1){
    //Extracts the lines between BeginOrderList and EndOrderList.
    ArrayList<String> temp = new ArrayList<>();
    line = br.readLine();
    while(!line.split("\\s+")[0].equals("EndOrderList")){
        temp.add(line);
        line = br.readLine();
        if(line == null){ throw new Exception("Error execFile() - No EndOrderList was
found."); }
    }
    //Creates an order list with the extracted lines then adds it to the array list of all
order lists.
    allOrderLists.addOrderList(addOrderList(temp));
    continue;
}
```

If the first argument of the line matches “BeginRestaurant” and the number of arguments is valid, the function extracts the name and delivery type of the restaurant form the first line and extracts the lines between the “BeginRestaurant” and “EndRestaurant” into an array list of strings where each string is a line. If the “EndRestaurant” is not found, an exception is thrown. The ‘addRestaurant()’ function is used to take the array list of lines and return a restaurant object which is then added to the array list of restaurant objects. The function then re-iterates and the next line in the file is read. Below is part of the implementation of the execFile() function where a new restaurant is added.

```
//Creating a new restaurant.
    if(tokens[0].equals("BeginRestaurant") && tokens.length == 3){
        //Extracts the restaurant name, delivery type and lines between BeginRestaurant and
EndRestaurant.
        String name = tokens[1];
        String delivery = tokens [2];
        ArrayList<String> temp = new ArrayList<>();
        line = br.readLine();
        while(!line.split("\\s+")[0].equals("EndRestaurant")){
            temp.add(line);
            line = br.readLine();
            if(line == null){ throw new Exception("Error execFile() - No EndRestaurant was
found."); }
        }
        //Creates a new restaurant with extracted lines then adds it to the array list of all
restaurants.
        restaurants.add(addRestaurant(name,delivery,temp));
        continue;
    }

    throw new Exception("Error execFile() - Command '"+line+"' not Recognized.");
}
```

At the end of the 'execFile()' function when the entire file is parsed and executed, the 'display()' function of the 'allOrderLists()' function is called which in turn notifies it's concrete observer to display the highest profited restaurant in the overall order lists as well as overall money spent.

```
}catch (Exception e){
        e.printStackTrace();
    }
    //Displays the highest revenue restaurant and overall spent on all the order lists.
    allOrderLists.display();
}
```

addOrderList() function

The following functions accepts an array list of string commands and returns an OrderList object. It starts by creating a new empty OrderList object and attaching it to a new OrderListObserver. Next, each line in the command array list is accessed in a for loop and a new array list of strings is created to store the lines between "BeginOrder" and "EndOrder". If a command is an empty line it is ignored, and the loop reiterates. Otherwise, the command is split into an array of string tokens using the space character as a delimiter. If the first argument in the command is "BeginOrder" and the number of arguments is valid, the restaurant name and delivery of the order is extracted as well as the lines between "BegindOrder" and "EndOrder" and stored in the new array list earlier created. If "EndOrder" was not found, an error is created. Below is part of the 'addOrderList()' function where an order is extracted from the file.

```
//Creates and returns an order list with a given array list of commands.
private OrderList addOrderList(ArrayList<String> input) throws Exception{
    //Creates a new order list and attaches an observer.
    OrderList orderList = new OrderList();
    new OrderListObserver(orderList);

    //Goes through each line of given input and creates orders.
    for (int i=0; i<input.size(); i++){
        ArrayList<String> order = new ArrayList<>();
        //Ignores empty lines.
        if(input.get(i).length() == 0){ continue; }
        String[] tokens = input.get(i).split("\\s+");



        //Creating a Order.
        if(tokens[0].equals("BeginOrder") && tokens.length == 3){
            i++;
            //Extracts the restaurant name, delivery type and lines between BeginOrder and
```

```
EndOrder.
            String name = tokens[1];
            String delivery = tokens[2];
            while(!input.get(i).split("\\s+")[0].equals("EndOrder")){
                if(i == input.size()-1){
                    System.out.println("Error addOrderList() - No EndOrder was found.");
                    System.exit(0);
                }
                order.add(input.get(i));
                i++;
            }
```

Next, the function searches in a restaurant array list for a restaurant matching the order using the 'findRestaurant()' function. A variable is made to keep track of the money spent in the current order. Next, the function goes through each element in the array list storing the lines between "BeginOrder" and "EndOrder" and adds the items name to a new array list and gets the cost of the item using the found restaurant's 'getPrice()' member function to add it to the profit variable. If a line does not begin with "OrderItem" and is not empty, the function will throw an exception stating the command could not be recognized. Below is part of the 'addOrderList()' function where the item names of an order were extracted from the file and their cost was found using a restaurant object that matched the order and appended to a float variable.

```
//Finds a restaurant matching the current order.
Restaurant restaurant = findRestaurant(name,delivery);
float profit = 0f; //Keeps track of money spent in the current order.

//Parses and executes the lines between BeginOrder and EndOrder (the item orders).
ArrayList<String> items = new ArrayList<>();
for(String item : order){
    if(item.length() == 0){ continue; }
    String[] token = item.split("\\s+");
    if(token[0].equals("OrderItem") && token.length == 2){
        //Adds the item name to an array list and adds its price to the profits made in the
current order.
        items.add(token[1]);
        profit += restaurant.getPrice(token[1]);
        continue;
    }
    throw new Exception("Error addOrderList() - Command '" + order + "' not recognized.");
}
```

Next the function adds the overall profit made in the current order, and then adds a new order using the name, delivery type, list of item names, profit made and matching restaurant object. The new order is then appended to the current order list object (which will update the order list observer). At the end of the 'addOrderList()' function the order list object is returned. Below is the end of the 'addOrderList()' function where an order is added to the order list.

```
            //Adds the overall profits of the current order to the matching restaurant.
            restaurant.addProfit(profit);
            //Adds the current order to the order list.
            orderList.addOrder(name,delivery,items,profit,restaurant);
            continue;
        }
        throw new Exception("Error addOrderList() - Command '" + input.get(i) + "' not
recognized.");
    }
    return orderList;
}
```

findRestaurant() function

The following function finds a restaurant that matches an order when given a name and delivery type. It works by looping through each restaurant in the global restaurant object array list and find a restaurant which matches the name and the delivery type (or a restaurant that matches the name and has both delivery types). If a restaurant is found, the found restaurant object is returned. Otherwise an exception is thrown saying the restaurant does not exist. Below is the implementation of the 'findRestaurant()' function.

```
    private Restaurant findRestaurant(String name, String delivery) throws Exception{
        for(Restaurant restaurant : restaurants){
            if(restaurant.getRestaurantName() == null){continue;}
            if(restaurant.getRestaurantName().equals(name) &&
(restaurant.getDeliveryType().equals(delivery) ||
restaurant.getDeliveryType().equals("both"))){
                return restaurant;
            }
        }
        throw new Exception("Error findRestaurant() - Restaurant with name '"+ name +"' and
delivery type '"+ delivery +"' was not found.");
    }
}
```

addRestaurant() function

The following function creates and returns a restaurant object given a name, delivery type and array list of strings representing the properties of an item that can be ordered. The function starts by creating a new restaurant object with the given name and delivery type. It then goes through each line the array list of items (skipping empty lines), and splits the line into token arguments using the space character (' ') as a delimiter. If the first argument of the line is "Item" and has the valid amount of arguments, the item name and item price (converted into a float) is added to the restaurant object's array list of items. If one of the commands do not start with the argument "Item", an exception is thrown. At the end of the 'addRestaurant()' function, the newly created restaurant has a populated item list and is returned. Below is the implementation of the 'addRestaurant()' function.

```
//Creates and returns a restaurant object with a given name, delivery type and a string list
of items.
private Restaurant addRestaurant(String name, String delivery, ArrayList<String> items) throws
Exception{
    //Create a temporary new restaurant object.
    Restaurant restaurant = new Restaurant(name,delivery);

    //Parses each line of the string array list of items and adds them to the restaurant item
list.
    for(String item : items){
        //Ignores empty lines.
        if(item.length() == 0){ continue; }

        String[] tokens = item.split("\\s+");
        if(tokens[0].equals("Item") && tokens.length == 3) {
            //Adds a new item to the restaurant with a name and price.
            restaurant.addItem(tokens[1], Float.parseFloat(tokens[2]));
            continue;
        }
        throw new Exception("Error addRestaurant() - Command '" + item + "' not recognized.");
    }
    return restaurant;
}
```

Order.java

The following file has the class definition of a single order whose objects will be stored as nodes in a linked list manner within an Orderlist object. Each order stores another order object which represents the order that follows the current one when it is added to the order list. The class variables include strings to store the name of the restaurant the order is for, and the delivery method. Each order will include an array list of strings representing item names and another class variable that stores a restaurant object. There is also a float variable to keep track of the total money spent in the current order. The constructor instantiates a new order list object and sets all the class variables with the given arguments. The rest of the class includes getter functions for the class variables. Below is the code used to define the Order class.

```
//A single Order/node in the OrderList linked list.
public class Order {
    Order next = null;
    private String restaurantName;
    private String deliveryType;
    private ArrayList<String> items;
    private Restaurant restaurant;
    private float total;

    Order(String restaurantName, String deliveryType , ArrayList<String> items, float total,
Restaurant restaurant){
        this.restaurantName = restaurantName;
        this.deliveryType = deliveryType;
        this.items = items;
        this.total = total;
        this.restaurant = restaurant;
    }



    public Restaurant getRestaurant(){return restaurant;}
    public float getTotal(){return total;}
    public String getRestaurantName(){return restaurantName;}
    public String getDeliveryType(){return deliveryType;}
    ArrayList<String> getItems(){return items;}
}
```

Item.java

The following file has the class definition of a single item whose objects will be stored in a Restaurant's object item list. The class variables include a string to store the name of the item and a float variable to store the price of an item. The constructor sets the class variables and the rest of the class include getter functions. Below is the code used to define the Item class.

```
//Defines a single item in a Restaurant item list.
class Item {
    private String itemName;
    private float itemPrice;

    Item(String itemName, float itemPrice){
        this.itemName = itemName;
        this.itemPrice = itemPrice;
    }
    String getItemName() { return itemName; }
    float getItemPrice() { return itemPrice; }
}
```

Restaurant.java

The following file has the class definition of a single restaurant. The class variables include strings to store the name of the restaurant and the delivery type (takeaway, delivery or both) and a float variable to store the profit of the restaurant. It also has an array list of Item Objects. The constructor sets the name and delivery type of restaurant and instantiates an empty item list and sets the profit to 0. It is important to note that an array list was used for its dynamic storage abilities. In order to populate the list, the 'addItem()' function is used which adds a new item object (defined by the Item class) given a name and price for the item.

The 'addProfit()' function is used to append a given float amount to the current profit of the restaurant. There is also a 'getPrice()' function which returns the price of an item given the name for the item to be found. The function loops through each item in the restaurants item array list and if one of the item matches the given name, the price of the item is returned. If no items in the restaurants item array list match the given name, an exception is thrown. Below is the code used to define the Restaurant class.

```
public class Restaurant {
    private String restaurantName;
    private String deliveryType;
    private float profit;
    private ArrayList<Item> items;

    public Restaurant(String restaurantName, String deliveryType){
        this.restaurantName = restaurantName;
        this.deliveryType = deliveryType;
        items = new ArrayList<>();
        profit = 0f;
    }

    //Returns a price for a particular item.
    public float getPrice(String name) throws Exception{
        for(Item item :items) {
            if (item.getItemName().equals(name)) {
                return item.getItemPrice();
            }
        }
        throw new Exception("Error getprice() - Item "+name+" can't be found in " +
restaurantName + ".");
    }

    public void addItem(String itemName, float itemPrice){ items.add(new
Item(itemName,itemPrice)); }
    public ArrayList<Item> getItems() { return items; }
    public void addProfit(float amount){ profit += amount; }
    public float getProfit(){ return profit; }
    public String getRestaurantName(){return restaurantName;}
    public String getDeliveryType(){return deliveryType;}
}
```

OrderList.java

The following file has the class definition of an order list. The order list stores multiple orders in a linked list manner. Each order is considered as node in the linked list. The class has a variable to keep track of the head of the list as well as the end.

The class has an observer object which will keep track of the overall money in the current order list as well as the highest profited restaurant in the current order list. Since the order list class is a subject class of the observer design pattern, it includes a function to attach an observer object to itself (specifically an OrderListObserver object). The order list class has functions that retrieve

information from the observer object such as the total money spent in the current order list, or the highest profit restaurant in the current order list. The 'display()' function calls the 'showInfo()' method of the observer object which outputs these monitored variables to the console.

The constructor of the class creates a new order list object and sets its head to an empty order. The 'addOrder()' function is used to populate the order list with orders. It functions by creating a new order object given a restaurant name, delivery type, array list of item names, order total and restaurant object. Next, the function searches for an available position in the order list for the new order. If the order at the head of the order list has empty content, it means that the order list is empty, so the new order is set as the head and end of the order list. If the head of the order list is not empty, the function loops through each order in the order list until the next order is empty. Then it sets next order to the new order and sets the end of the order list to the new order. Every time an order is added to the order list, the observer object update function is called. Below is the code used to define the OrderList class.

```
public class OrderList {
    private Order head;
    private Order end;
    private Observer observer;

    public OrderList(){
        head = new Order(null, null, null,0f , null);
    }

    //Adds a new Order/Node to the current Order List/Linked List and updates the observer.
    public void addOrder(String restaurantName, String deliveryType, ArrayList<String> items,
float total, Restaurant restaurant){
        Order newOrder = new Order(restaurantName,deliveryType,items,total,restaurant);
        //If the head is null, add the order to the start of the order list.
        if(head.getRestaurantName() == null && head.getDeliveryType() == null &&
head.getItems() == null){
            head = newOrder;
            end = newOrder;
            observer.update();
            return;
        }
        //Else add the order to the end of the order list.
        Order current = head;
        while(current.next != null){
            current = current.next;
        }
        current.next = newOrder;
        end = newOrder;
        observer.update();
    }

    //Returns the last Order/Node.
    public Order getEnd(){return end;}
    //Attaching an observer and calling observer functions.
    public void attach(Observer observer){ this.observer = observer;}
    public void display(){observer.showInfo();}
    public float getTotal(){ return observer.getTotal();}
    public Restaurant getHighestProfit(){return observer.getHighestProfit();}
}
```

AllOrderLists.java

The following file has the class definition of an object that stores all the order lists. The class has an array list of OrderList objects and keeps an OrderList object track of the last order list added. This class was created to keep track of all the order lists that have been added and considered when parsing the file.

The class has an observer object which will keep track of the overall money spent in all the order lists as well as the most profited restaurant. Since the class is a subject class of the observer design pattern, it includes a function to attach an observer object to itself (specifically an AllOrderListsObserver object). The class has functions that retrieve information from the observer object such as the total money spent, or the highest profit restaurant in all the order lists. The 'display()' function calls the 'showInfo()' method of the observer object which outputs these monitored variables to the console.

When created, the array list of order lists is empty. To populate the array list, the 'addOrderList()' function is used, which simply appends a given OrderList object to the array list and updates the last order list variable. Every time a new order list is added the update function of the observer function. Below is the code used to define the AllOrderLists class.

```
public class AllOrderLists {
    private ArrayList<OrderList> orderLists;
    private OrderList lastOrderList;
    private Observer observer;

    public AllOrderLists(){
        orderLists = new ArrayList<>();
    }

    //Adds a new OrderList and updates the observer.
    public void addOrderList(OrderList orderList){
        orderLists.add(orderList);
        lastOrderList = orderList;
        observer.update();
    }

    //Returns the last OrderList.
    public OrderList getLastOrderList() { return lastOrderList; }
    //Attaching an observer and calling observer functions.
    public void attach(Observer observer){ this.observer = observer; }
    public void display(){observer.showInfo();}
}
```

Observer.java

The following file defines the Observer class which is used as part of the implementation of the Observer Design Pattern in the program. The Observer class is used to define the operations to be used in the concrete observe classes (found in OrderListObserver.java and AllOrderListsObserver.java). The class is an abstract class since the concrete implementation of the methods will be present in the observer's subclasses.

The observer class includes function to update the observer's variables ('update()'), to display the observers variables ('showInfo()'), and to get the total money spent or highest profited restaurant in the observer ('getTotal() and getHighestProfit()'). The observer subclasses will keep track of the total money spent and the highest profited restaurant for each order list or all the order lists combined. Below is the code used to define the Observer class.

```
public abstract class Observer {
    //Update the contents of the observer object.
    public abstract void update();
    //Display the contents of the observer object.
    public abstract void showInfo();
    //Return the total spent and highest revenue restaurant of the observer object.
    public abstract Restaurant getHighestProfit();
    public abstract float getTotal();
}
```

OrderListObserver.java

The following file defines the concrete observer class OrderListObserver which keeps track of the amount spent and highest profited restaurant in a single OrderList object. The class has a float variable to keep track of the total spent, as well as a restaurant object which will store the highest profited restaurant in the order list. The class also stores an OrderList object representing the OrderList it's observing. The constructor instantiates the observer and assigns a given order list to its OrderList object and attaches itself to the Observer Object of the given order list. It also declares the highest profited restaurant object as an empty one.

When the concrete observer's 'update()' method is called (after an order is added to an order list), the observer obtains the last order from the order list and appends the money spent in the latest order to the tracked total money spent. In the ReadFile class (defined in ReadFile.java) , the restaurant profit of an order is appended when an order is added to an order list object. Hence, the observer update function checks whether the restaurant of the latest order is now higher than the current tracked highest profit restaurant. If this condition is true, the restaurant of the latest order is set to the highest profited restaurant of the current observer object.

When the 'showInfo()' method is called, the concrete observer displays the name of the current highest profited restaurant in the current order list as well as its overall profit made to the console. It also displays the overall money spent in the current order list. The class also includes two functions to return the highest profited restaurant object and the total money spent. Below is the code used to define the OrderListObserver class.

```
public class OrderListObserver extends Observer {
    private float total;
    private Restaurant highestProfit;
    private OrderList orderList;

    public OrderListObserver(OrderList orderList){
        this.orderList = orderList;
        this.orderList.attach(this);
        this.highestProfit = new Restaurant(null,null);
    }

    @Override
    public void update(){
        //Obtains the latest added order.
        Order newOrder = orderList.getEnd();
        Restaurant newOrderRestaurant = newOrder.getRestaurant();
        //Updates the total and the highest revenue restaurant (if applicable) for current
order list.
        total += newOrder.getTotal();
        if(newOrderRestaurant.getProfit() > highestProfit.getProfit()){
            highestProfit = newOrderRestaurant;
        }
    }

    @Override
    public void showInfo(){
        //Displays the the overall spent and the highest revenue restaurant for the current
order list.
        System.out.println("The highest revenue restaurant is '" +
highestProfit.getRestaurantName() + "' with a total revenue of
"+highestProfit.getProfit()+"€.");
        System.out.println("The total price all order lists is " + total + "€.");
    }

    @Override
    public float getTotal(){return total;}

    @Override
    public Restaurant getHighestProfit(){return highestProfit;}
}
```

AllOrderListsObserver.java

The following file defines the concrete observer class OrderListObserver which keeps track of the amount spent and highest profited restaurant in a single AllOrderLists object. The class is implemented in the exact same way as the OrderListObserver class except the subject of this class is an AllOrderList object instead of an OrderList object. Hence, the constructor instantiates the observer and assigns a given AllOrderLists object to its own AllOrderLists object and attaches itself to the Observer Object of the given AllOrderLists object. It also declares the highest profited restaurant object as an empty one.

Hence, the 'update()' method of this class is called when a new order list is added to an AllOrderList object. The function obtains the latest added order list and appends its total to the current observer's total money spent variable. It also checks whether the highest profited restaurant's profit is higher than the observer's current highest profited restaurant. If this condition is true, the highest profited restaurant of the newly order list is set as the new highest profited restaurant in the observer. It is important to note that each time the update method is called, the total of the newly added order list is displayed.

When the 'showInfo()' method is called, the concrete observer displays the name of the highest profited restaurant in the current AllOrderList object as well as its overall profit made to the console. It also displays the overall money spent in the overall order lists. The class also includes two functions to return the highest profited restaurant object and the total money spent. Below is the code used to define the AllOrderListsObserver class.

```
public class AllOrderListsObserver extends Observer {
    private float total = 0;
    private Restaurant highestProfit;
    private AllOrderLists orderLists;

    public AllOrderListsObserver(AllOrderLists OrderLists){
        this.orderLists = OrderLists;
        this.orderLists.attach(this);
        this.highestProfit = new Restaurant(null,null);
    }

    @Override
    public void update(){
        //Obtains the latest added order list.
        OrderList newOrderList = orderLists.getLastOrderList();
        Restaurant newOrderHighest = newOrderList.getHighestProfit();
        //Displays the information of the latest order list.
        System.out.println("Added a new order list with a total of: " + newOrderList.getTotal() +
"€.");
        //Updates the highest revenue restaurant (if applicable) and the overall spent.
        total += newOrderList.getTotal();
        if(newOrderHighest.getProfit() > highestProfit.getProfit()){
            highestProfit = newOrderHighest;
        }
    }

    @Override
    public void showInfo() {
        //Displays the the overall price of all order lists and the restaurant with the highest
profit.
        System.out.println("-------------------------------------------------------------------------
--");
        System.out.println("The highest revenue restaurant is '"+highestProfit.getRestaurantName() +
"' with a total revenue of "+highestProfit.getProfit()+"€.");
        System.out.println("The total price all order lists is "+total+"€.");
    }

    @Override
    public float getTotal(){return total;}

    @Override
    public Restaurant getHighestProfit(){return highestProfit;}
}
```

Bash Scripts (.sh)

compile.sh

This file compiles all the .java files which are found in the src folder into bytecode. Below is a screenshot of the contents of the file.

```
#!/bin/bash

javac src/Runner.java src/ReadFile.java src/Object/AllOrderLists.java src/
Object/Item.java src/Object/Order.java src/Object/OrderList.java src/Object/
Restaurant.java src/Observer/AllOrderListsObserver.java src/Observer/
Observer.java src/Observer/OrderListObserver.java
```

run.sh

This file executes the application by calling the class containing the main method and specifying the file path to be used by checking whether an additional argument was passed when running the bash script. If the user adds an additional argument, the script executes the program using the added argument as the file path. If no arguments were provided, the path of the test file is used when running the program. Below is a screenshot of the contents of the file.

```
#!/bin/bash

cd src
#If additional argument wasn't given, run the default test file.
if [ "$#" -eq "0" ]
then
    java Runner "../test.txt"
else
    java Runner "$1"
fi
cd ..
```