Middle East Technical University | Department of Computer Engineering

CENG 213

Data Structures

Fall 2012-2013

Homework 2 - Graphical User Interface

Due date: 16 December 2012, Sunday, 23:55

1 Objective

This homework aims to help you get familiar with the famous abstract data type: *Tree*. You will implement a ridiculously simplified graphical user interface (GUI) system using C++.

Keywords: *Object Oriented Programming, C++, Tree*

2 Problem Definition

In this homework, you are asked to implement a basic graphical user interface system that stores user components in a tree. In our simplified setting, a component is represented as a rectangle by specifying its *name*, *position* and *size*. Each component has a parent (except for the root component) and may contain a number of child components. As you may have guessed, this structure is known as *Tree* in computer science. In short, you are supposed to implement two classes: `Component` and `GUI`.

3 Component

This is a simple class that represents a component in our graphical user interface system. As mentioned above, a component has a *name* (`std::string`), *position* (`std::pair<int,int>`) and *size* (`std::pair<int,int>`). It also contains a pointer to its parent component/container (`Component*`) and maintains a list of child components (a list of `Component*`). The header file for this class, named "Component.h", is listed below. You are expected to implement the methods documented in this header file. You are not allowed to change the prototoypes of existing methods but you are welcome to modify the header by adding extra methods or fields.

```
#ifndef __COMPONENT_H__
#define __COMPONENT_H__
#include <string>
#include <vector>

using namespace std;

typedef pair<int,int> Position;
```

```
typedef pair<int,int> Size;

class Component
{
public:
  /**
   * Constructs a Component object using given parameters. Initially,
   * a component has no child child components and its parent is set
   * to NULL.
   *
   * @name[in]     a string that specifies the name of the component
   * @size[in]     std::pair<int,int> that specifies the size/dimensions
   *               of the component
   * @position[in] std::pair<int,int> that specifies the position of
   *               the component
   */
  Component(string name, Size size, Position position);
  /**
   * Gets/sets the name of the component.
   */
  string               getName();
  void                 setName(string name);
  /**
   * Gets/sets the size/dimensions of the component.
   */
  Size                 getSize();
  void                 setSize(Size size);
  /**
   * Gets/sets the positions of the component.
   */
  Position             getPosition();
  void                 setPosition(Position pos);
  /**
   * Gets/sets the parent of the component. If this is the root
   * component, "getParent" returns NULL.
   */
  Component*           getParent();
  void                 setParent(Component* component);
  /**
   * Adds a new component as a child of this component. This method
   * does not check whether the child component is enclosed by
   * this component. This is the job of the GUI::insertComponent.
   */
  void                 addChild(Component* child);
  /**
   * Returns a pointer to the component having the name specified
   * by the argument.
   */
  Component*           getChild(string name);
  /**
   * Removes the component having the name specified by the argument.
   * It does not delete/free the removed component; it only removes the
   * specified component from the children list.
   *
   * @name  the name of the component to be removed
   * return true if the component exists and removed, false otherwise
   */
```

```
  bool                  removeChild(string name);
  /**
   * Returns all child components as a vector. Note: it only returns
   * its children, not grand-children or lower descendents.
   *
   * return a reference to a vector of Component pointers
   */
  vector<Component*>&  getChildren();
  // TODO: add methods if required
private:
  // TODO: add methods/fields if required
};
#endif
```

4 GUI

This is the core class of the system. It allows users to add/remove components and display existing components. The GUI is represented as a two-dimensional grid. The origin of the grid is the upper left corner (see Figure 1). A position in this grid is specified using a typedef defined in "Component.h": `Position`. This is actually equivalent to `std::pair<int,int>` class, which contains two int fields: `first` and `second`. The location (i, j) is, thus, represented as `Position(i,j)`. In other words, `std::pair::first` specifies the index in the vertical direction and `std::pair::second` specifies the index in the horizontal direction.

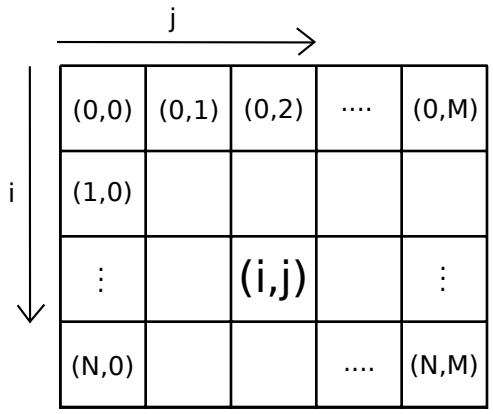


Figure 1: Grid layout used to position components. Here we see a grid of size $N \times M$. This grid is constructed using the following code: `Grid(Size(N,M))` where `Size` is defined as `std::pair<int,int>` (`std::pair::first` specifies the height of the grid whereas `std::pair::second` specifies the width).

4.1 Basic Functionality

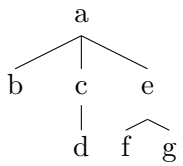
The GUI contains a default component that is called the "root" component. The name of this component is "root" and it has the same size as the GUI and positioned at $(0, 0)$. You may assume that the root component will not be removed from the GUI and it will be the ancestor of other components in the GUI. Components are added to the GUI via the void `insertComponent(Component* component)` method. This method inserts a given component as a child of the *nearest* and *deepest* component in the GUI. In other words, you need to find the deepest component which covers the upper left corner of the given component. Note that the parent component does not need to completely cover the given component; it suffices to cover only the upper left corner point. If there are more than one component that satisfy this criteria you may choose one of them arbitrarily as the parent.

The GUI class has a method called `std::vector< std::pair<string,int> > areaUsage()`, which computes cumulative area usage of each component in the GUI. This functionality is similar to the famous disk usage command in Linux: `du`, which recursively produces a list of files with disk usage information. Our method returns a `std::vector` of `std::pair<string,int>` that specifies the name and total area of each component. Since components are rectangles in our setting, area computation is simple. The total area of a component is equal to the sum of its own area and the areas of its children recursively. Note that you do not need to subtract areas shared among components; it is sufficient to compute the area of each component and its descendents recursively. Note that components will be inserted to the vector in post-order.

4.2 Rendering

The GUI is rendered in two different ways:

- *Hierarchical form*: This functionality is provided via the overloaded left-shift operator: `std::ostream& operator<<(std::ostream& os, const GUI& gui)`. The component hierarchy is displayed in a specific format in which each level is printed on a separate line and children of each node are enclosed with '|' characters and separated by the blank character. For instance, given the following tree,

  the hierarchical form renders/prints the GUI as follows:

  ```
  | a |
  | b c e |
  | | | d | | f g |
  | | | | | | |
  ```

  Extra space at the end of each line is not important but you are supposed to put spaces between each children and '|' characters.

- *2-D scene form*: This functionality is exposed via a method called `int** toArray()`, which returns an $N \times M$ matrix where N is the height and M is the width of the GUI. These parameters are set by the constructor of the GUI class: `GUI(Size size)`. This matrix contains positive numbers for edges and zero for non-edges (surface). You are supposed to print components in a hiearchical way according to their depth in the component tree. In other words, components that are at higher levels of the tree should be occluded by components that are at lower levels. If two components have the same depth, regardless of they are siblings or not, they should be displayed transparent to each other, i.e., they should not occlude each other. You are recommended to take a look at the sample code and its output shared on COW.

4.3 Header File

You are supposed to implement all the methods listed in the header file named "GUI.h", which is listed below. You are free to add extra methods or fields if required but you should not change the prototypes of existing methods.

```
#ifndef __GUI_H__
#define __GUI_H__
#include <iostream>
#include <string>
#include <vector>
#include "Component.h"

using namespace std;

class GUI
{
public:
  /**
   * Constructs a GUI object with the given @size (dimensions).
   *
   * @size[in] an instance of std::pair<int,int> that specifies the
   *           dimensions of the graphical user interface (GUI)
   */
  GUI(Size size);
  /**
   * @return the root Component of the Component tree (hierarchy)
   */
  Component*                  getRoot();
  /**
   * @return the size/dimensions of the root Component
   */
  Size                        getSize();
  /**
   * This method inserts a new component to the GUI. The @component
   * is inserted under the deepest component that covers the upper
   * left corner of the argument @component. If there are more than
   * one such component that satisfy this criteria, choose one
   * arbitrarily.
   *
   * @component[in] a pointer to a Component object
   */
  void                        insertComponent(Component* component);
  /**
   * This method removes the component called @name and all its children
   * recursively. Note: "root" component will never be removed.
   *
   * @name[in] the name of the component
   * @return   true if the component exists (and removed), false otherwise
   */
  bool                        removeComponent(string name);
  /**
   * This method returns the deepest component that contains the given @point.
   * In case there are more than one such component return one of them
   * arbitrarily.
   *
   * @point[in] an instance of std::pair<int,int> that specifies a point
   * @return    a pointer to target Component
   */
  Component*                  contains(Position point);
  /**
   * This method computes cumulative area usage of each component in the GUI.
   * Since each component is a rectangle, the area is computed using their
```

```
   * width and height. In addition, each component sums up the area usage of
   * its children recursively. This is similar to the "du" command in Linux.
   * Components will be inserted to the vector in post-order.
   *
   * return a vector containing component name, total area pairs
   */
  vector< pair<string,int> >  areaUsage();
  /**
   * This method returns a matrix that displays the components as a 2-D scene.
   * This matrix contains positive numbers at a location P if P contains an edge.
   * Note that components that are at lower levels (i.e., deeper in the
   * hierarchy) occlude components that are at higher levels.
   *
   * @return a matrix as a two-dimensional jagged array
   */
  int**                       toArray();
  /**
   * This overloaded operator<< prints the GUI component tree in a specific
   * format in which each level is printed on a separate line and children of
   * each node is enclosed with '|' characters. For instance, the following tree,
   *      a
   *    /   \
   *  b      c
   * is printed as:
   * | a |
   * | b c |
   * | | | |
   * Also, the following tree,
   *       a
   *    /\   \
   *  b  c    d
   *          |
   *          e
   * is printed as:
   * | a |
   * | b c d |
   * | | | | | e |
   * | |
   */
  friend ostream&             operator<<(ostream& os, const GUI& gui);
  // TODO: add methods if required
private:
  // TODO: add fields/methods if required
};

#endif
```

5 Sample Code

Here is a sample code that demostrates the usage of `Component` and `GUI` classes.

```
#include "Component.h"
#include "GUI.h"
#include <iostream>

using namespace std;
```

```
/**
 * This function prints the area usage for each component
 * using the GUI::areaUsage() method.
 */
void printAU(ostream& os, GUI& gui)
{
  os << "----------" << endl;
  vector< pair<string,int> > areas = gui.areaUsage();
  for (int i=0; i < areas.size(); ++i) {
    pair<string,int> node = areas[i];
    os << areas[i].first << " " << areas[i].second << endl;
  }
}

/**
 * This function prints the GUI as a two dimensional grid using the
 * GUI::toArray() method. It prints a star character ('*') for each
 * positive number of the array.
 */
void print2D(ostream& os, GUI& gui)
{
  int** array = gui.toArray();
  Size size = gui.getSize();
  for (int i=0; i < size.first; i++) {
    for (int j=0; j < size.second; j++) {
      if (array[i][j] == 0) {
        os << ' ';
      } else {
        os << '*';
      }
    }
    os << endl;
  }
}

/* Sample code block - 1 */
int main1()
{
  GUI gui(Size(10,10));
  Component* c;
  c = new Component("panel1", Size(4,5), Position(2,2));
  gui.insertComponent(c);
  printAU(cout, gui);  // Area-usage
  cout << gui;         // Hierarchical form
  cout << "----------" << endl;
  print2D(cout, gui);  // 2-D scene form
  return 0;
}
/* Sample code block - 2 */
int main2()
{
  GUI gui(Size(30,30));
  Component* c;
  c = new Component("panel1", Size(10,10), Position(5,5));
  gui.insertComponent(c);
  c = new Component("panel2", Size(10,10), Position(15,15));
```

```
  gui.insertComponent(c);
  c = new Component("button1", Size(5,5), Position(7,7));
  gui.insertComponent(c);
  printAU(cout, gui);  // Area-usage
  cout << gui;         // Hierarchical form
  cout << "----------" << endl;
  print2D(cout, gui);  // 2-D scene form
  return 0;
}
/* Sample code block - 3 */
int main3()
{
  GUI gui(Size(30,30));
  Component* c;
  c = new Component("panel1", Size(20,20), Position(5,5));
  gui.insertComponent(c);
  c = new Component("button1", Size(10,5), Position(10,10));
  gui.insertComponent(c);
  c = new Component("label1", Size(10,10), Position(20,20));
  gui.insertComponent(c);
  c = new Component("panel2", Size(4,8), Position(20,0));
  gui.insertComponent(c);
  c = new Component("image1", Size(5,10), Position(12,15));
  gui.insertComponent(c);
  printAU(cout, gui);  // Area-usage
  cout << gui;         // Hierarchical form
  cout << "----------" << endl;
  print2D(cout, gui);  // 2-D scene form
  return 0;
}
/* Sample code block - 4 */
int main4()
{
  GUI gui(Size(80,80));
  Component* c;
  c = new Component("panel1", Size(20,20), Position(20,20));
  gui.insertComponent(c);
  c = new Component("button1", Size(10,5), Position(25,25));
  gui.insertComponent(c);
  c = new Component("label1", Size(30,30), Position(40,40));
  gui.insertComponent(c);
  c = new Component("panel2", Size(60,10), Position(50,50));
  gui.insertComponent(c);
  c = new Component("button2", Size(5,5), Position(22,22));
  gui.insertComponent(c);
  c = new Component("radio1", Size(2,2), Position(23,23));
  gui.insertComponent(c);
  c = new Component("panel3", Size(40,40), Position(70,70));
  gui.insertComponent(c);
  printAU(cout, gui);  // Area-usage
  cout << gui;         // Hierarchical form
  cout << "----------" << endl;
  print2D(cout, gui);  // 2-D scene form
  return 0;
}
/* Main entry point of the application */
int main(int argc, char*argv[])
```

```
{
  char select = '1';
  if (argc == 2) {
    select = argv[1][0];
  }
  switch (select) {
  case '1':
    return main1();
    break;
  case '2':
    return main2();
  case '3':
    return main3();
  case '4':
    return main4();
  default:
    return main1();
  }
  return 0;
}
```

When run without any argument, the output of this code is as follows. You can reach the output of other code blocks on COW. **Note:** This sample code has several memory leaks because instances of the `Component` class are dynamically allocated but not deleted at the end. It is **not** your responsibility to analyse or correct these leaks (Neither the `GUI` class nor the `Component` class needs destructors).

```
----------
panel1 20
root 120
| root |
| panel1 |
| |
----------
**********
*        *
* *****  *
* *   *  *
* *   *  *
* *****  *
*        *
*        *
*        *
**********
```

6 Regulations

- **Programming Language:** You must code your program in C++. Your submission will be compiled with `g++` on department lab machines. You are expected to make sure your code compiles successfully with `g++`.
- **Late Submission:** You have a total of 7 days for late submission. You can spend this credit for any of the assignments or distribute it for all. If total of late submissions exceeds the limit, a penalty of $5 * days * days$ is applied.

- **Cheating:** In case of cheating, the university regulations will be applied.
- **Newsgroup:** You must follow the newsgroup (news.ceng.metu.edu.tr) for discussions and possible updates on a daily basis.
- **Grading:** This homework will be graded out of 100. It will make up 10% of your total grade.

7 Submission

Submission will be done via COW. Create a tar.gz file named `hw2.tar.gz` that contains all your source code files (Component.h, Component.cpp, GUI.h, GUI.cpp). This archive should not contain a source file with a `main` function. Assuming that "sample.cpp" exists in the current directory, the following command sequence is expected to compile and run your program on department computers. You are encouraged to use the accompanying Makefile to compile your code.

```
$ tar -xf hw2.tar.gz
$ g++ sample.cpp Component.cpp GUI.cpp -o hw2
$ ./hw2
```