Name:_____**Registration-Nb.:**_____

1 Test Basics

1.1 What is the goal of Software Tests?

1.2 Describe three software quality criteria and describe which tests could be used to check them!

1.3 Why is prioritization important in testing?

1.4 Explain "White Box Testing", "Black Box Testing", and "Gray Box Testing"

1.5 What is the difference between a „Test Case“ and a „Test Scenario“?

1.6 What is the meaning of “severity level” and “priority” in defect management?

2 Test Strategy

2.1 What is the goal of the Test Strategy Phase?

2.2 Discuss the advantages and disadvantages concerning the assignment of the responsibility concerning test to

a. the Software vendor

b. the customer

c. Test specialists

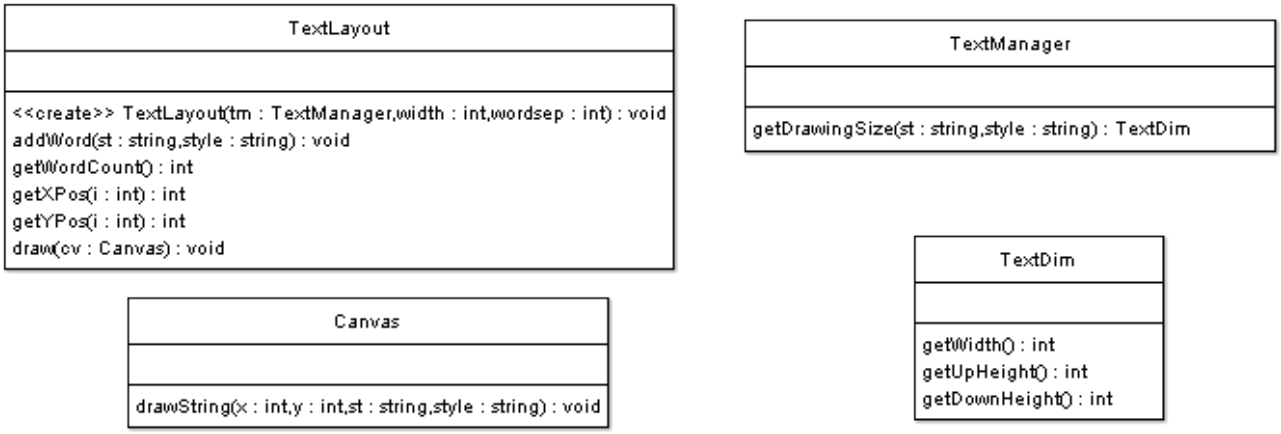
2.3 Describe possibilities to improve the test process!

3. Text layout

In this problem, you will write junit test cases for class `TextLayout` which takes a sequence of words and their display styles, and computes their positions in a drawing area. You can download `TextLayout.java` and a skeleton for `TextLayoutTest.java` from `http://garnet.cpe.ku.ac.th/~jtf/219343/`. This implementation of TextLayout contain some defect, and it will be great if your test case can reveal it.

You should send your solutions to problems 3.1 and 3.2 to jittat@gmail.com. Because they will have the same file name, you can rename the solution to problem 3.1 to `TextLayoutTest31.java`.

The following diagram shows relevant classes.

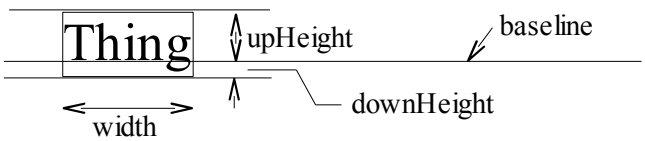


An object of class TextLayout is created with the following arguments: an object of class TextManager `tm`, drawing area `width`, and the space between words `wordsep`.

From the interface of `TextLayout`, you can see that it takes each word and its style incrementally (method `addWord`). An example of using this method is shown below.

```
TextLayout to = new TextLayout(tm,100,5);
to.addWord("Hello", "Normal");
to.addWord("World", "Bold");
```

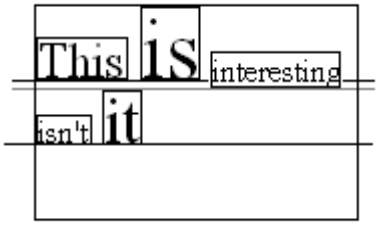
To get an information for computing layouts, `TextLayout` would call a method `getDrawingSize` from `TextManager`, passing `st` and `style` as parameters, to compute the dimension of that word; this method returns an object of `TextDim`. The dimension of a word is show below.



We now describe how class `TextLayout` find positions for words in the texts. The texts are divided into lines. Words on the same line are vertically aligned so that they are on the same baseline. `TextLayout` tries to fit as many words as possible in the same line, but make sure that the there is a space of `wordsep` between every pair of words. The upHeight (downHeight) of a line is the

maximum upHeight (downHeight) of words on that line.

After `TextLayout` receives each word, it recomputes the position of every word. The interface to layout information are methods `getWordCount`, which returns the number of words received so far; `getXPos`, and `getYPos`, which take the index to words (starting at 0) and return its x and y positions. The co-ordinate of the upper-left corner of the drawing area is (0,0). A sample layout is show below.



Note: The width of the containing box is specified as `width`, and the size of spaces between words is specified as `wordsep` in the constructor. All words in a single line share the same baseline. The top of each line (at upheight) matches the bottom (at downheight) of the previous line; see the word "it" in the picture.

3.1 Testing TextLayout, mocking TextManager

Since we want to test TextLayout for any possible TextManager, we decide that we will mock TextManager. (Note that TextManager is an interface.)

Write two interesting test cases for TextLayout (in class TextLayoutTest).

The following is a skeleton of `TextLayoutTest.java`.

```
import org.junit.Test;
import static org.junit.Assert.*;
import static org.easymock.EasyMock.*;

class TextDim {
    private int width, upheight, downheight;
    public TextDim(int w, int uh, int dh) {
           width = w;
           upheight = uh;
           downheight = dh;
    }
    public int getWidth() { return width; }
    public int getUpHeight() { return upheight; }
    public int getDownHeight() { return downheight; }
}

interface TextManager {
    TextDim getDrawingSize(String st, String style);
}

interface Canvas { // this Canvas interface will be used in problem 3.2
    void drawString(int x, int y, String st, String style);
}

public class TextLayoutTest {
    ....  // your tasks.
}
```

You must mock `TextManager` using EasyMock framework. Since `TextLayout` can call method `getDrawingSize` for each word many times, when programming your mock you must specify this behavior. You can do so by calling method `atLeastOnce` after calling `expect` as shown in an example below.

```
TextManager tm = createMock(TextManager.class);
expect(tm.getDrawingSize("Hello", "Normal"))
    .andReturn(new TextDim(20,10,0)).atLeastOnce();
expect(tm.getDrawingSize("World", "Bold"))
    .andReturn(new TextDim(25,12,0)).atLeastOnce();
```

You can use `getWordCount`, `getXPos`, and `getYPos` to verify that `TextLayout` works correctly.

Note: your test case shouldn't be too trivial or too complicated. You might want to read problem 3.2 before you continue.

3.2 Testing TextLayout, mocking both TextManager and Canvas

We want our testing paradigm to move to interaction-based testing. So, we decide to drop three state-checking methods `getWordCount`, `getXPos`, and `getYPos` from `TextLayout`. We still want to test the class, but to verify its correctness, we need to look at its interaction with class `Canvas`, which interacts with TextLayout through method `draw`. If you call method draw, passing in the `Canvas`, TextLayout would use method `drawString` of `Canvas` to draw each word, from the first to the last one.

Modify the test case in problem 3.1, so that it also mocks `Canvas` and the correctness of TextLayout is verified through its interaction with the mocked canvas.