Institutionen för Datavetenskap. Lunds Universitet. Dat501: OOP, VT03

Prov på delkurs Objekt-orienterad Programmering

Ordinarie tentamen 6. juni 2003

Tillåtna hjälpmedel: Litteratur, egna anteckningar

Tentamen består av 3 uppgifter, varje uppgift kan ge 10 poäng. Deluppgifternas poäng anges inom [hakparenteser]. Observera att poänggivningen inte nödvändigtvis avspeglar deluppgiftens svårhetsgrad. Uppgifterna kan besvaras på svenska eller engelska.

Behandla högst en uppgift per papper (det går bra att behandla deluppgifter på samma papper). Skriv bara på ena sidan och markera varje sida med dina initialer. Skriv läsligt.

▶ a)[0] Hur många uppgifter får man behandla per papper?

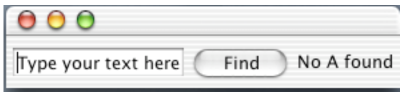
▶ b)[0] Får man skriva på baksidan?

Det går bra att referera till kursböckerna av Schmidt, Hansen och Rischel eller Rosen. Referenser måste ha formen [S: *något*], [HR: *något*] eller [R: *något*], där *något* är ett avsnittsnummer, en kodbit, en räknad uppgift, etc. Skriv till exempel "genom att använda metoden **validate** från [S: Figure 7 i Chapter 6]". Vill du referera till något annat, måste du ange bokens titel, författere, och förlag. Skriv till exempel "genom att använda metoden *validate* från [Barry Cornelius: Understanding Java, Addison–Wesley 2001, p. 254]".

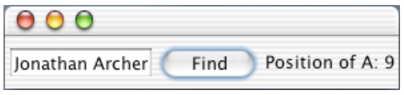
Ibland hänger vissa uppgifter ihop. Observera att det går bra att referera till svaret på tidigare uppgifter, även om du inte har löst dem.

Exercise 1 (10 points total)

Consider the following simple Swing application:



The user types some text into the JTextArea on the left and presses the 'Find' JButton. This will locate the letter 'A' in the text, as in the following example:



Here is the Java code for this application.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class FindLetterFrame extends JFrame implements ActionListener
{
  JTextField input_text;
  JButton find_button;
  JLabel found_text;

  public FindLetterFrame()
    {
      input_text= new JTextField("Type your text here");
      find_button= new JButton("Find");
      find_button.addActionListener(this);
      found_text= new JLabel("No A found");

      Container c= getContentPane();
      c.setLayout(new FlowLayout());
      c.add(input_text);
      c.add(find_button);
      c.add(found_text);

      pack();
      setVisible(true);
    }

  public void actionPerformed(ActionEvent e)
    {
      String S= input_text.getText();
      int pos= S.indexOf('A');
      found_text.setText("Position of A: "+pos);
    }

  public static void main(String[] args)
    { new FindLetterFrame(); }
}
```

▶ a)[2] Whenever the input text does not contain an 'A' the label should read 'No A found'. Change the code to achieve this.

We will change this application to follow the MCV paradigm. First we introduce a model to do the searching for us. The model will know about the character to be searched for ('A' in our example above, but it should be able to search for other characters instead). The specification is the following:

public class SearcherModel	
Constructor: public SearcherModel(char c)	Sets the searched-for character
Methods: public int search(String s)	Returns the position of the character in the string s, or -1 if the character doesn't appear

▶ b)[1] Implement SearcherModel.

▶ c)[2] Change FindLetterFrame so as to use a '`new SearcherModel('A')`' as its model.

▶ d)[2] Now separate the controller by extending the JButton.

▶ e)[2] Write a class diagram of your application.

Currently our application does case-sensitive searching (i.e., it treats 'A' and 'a' as different characters). We now what a second button (called 'Switch') that switches between case-sensitve and case-insensitive searching. So after pressing the Switch button once, the application now treats 'A' and 'a' as the same letter, no matter how many new texts are entered, or how often the Find button is used. Pressing the Switch button again reverts to the original meaning.[1]

▶ f)[2] Add the Switch button to your application and make the necessary changes. It is important that the information about which of the two searching states we are in (case-sensitive or case-insensitive) is kept in the *model*, not in the controller or the view.

Note: To avoid writing a lot of code it is fine to use 'three-dots notation' to quote the original code, instead of copying the corresponding lines of code by hand. For example:

```
input_text= ...
... // as in FindLetterFrame
... new FlowLayout());
```

[1]Please don't use a JToggleButton just to show how cool you are, even if that probably would produce a nicer user interface

Exercise 2 (10 points total)

Here is a function that is supposed to compute 2^x on input x:

```
/** Computes two-to-the-power of a given number.
 * @param $x \geq 1$ an integer
 * @return $2^x$.
 */
static int raise(int x)
{ int $i$=1;
{ int $r$=2;
  while ($i \neq x$)
        // loop invariant: $r = 2^i$
        { loop body }
  return $r$;
}
```

▶ a)[2] Write the loop body so that it satisfies the invariant (you don't have to prove that).

▶ b)[2] Assuming that the loop body satisfies the invariant, prove that raise is partially correct.[2]

▶ c)[1] Change raise so that it works with the (weaker) condition that x is an integer with $x \geq 0$.

The *integer logarithm base two* of x is defined so to be the integer y satisfying

$$2^y \leq x < 2^{y+1}.$$

For example, 3 is the integer logarithm of 8, and also of 10. (You can view this function as $y = \lfloor \log_2 x \rfloor$ if that makes it clearer for you.)

Consider the following function

```
/** Computes the integer logarithm of a given number.
 * @param $x$ a positive integer $\geq 1$
 * @return $\lfloor \log_2 x \rfloor$.
 */
static int ilog(int x)
{ int $y$= 1;
  while (raise($y$) < $x$)
        $y$= $y$+1;
  return $y$;
}
```

▶ d)[1] (Very easy) Find the integer logarithm of 4 and of 5.

▶ e)[4] (Harder) Prove that ilog is partially correct (this includes fining a valid invariant), and explain briefly why it terminates.

[2]Note that this exercise *doesn't* ask you to prove that the loop body satisfies the invariant. Please don't waste time doing that.

Exercise 3 (5 points total)

Consider the following declarations:

```
public interface Animal
 { public String speak(); }

public class Dog implements Animal
 { public Dog() { }
   public String speak() { return "Grrr!"; }
   public void eat() { }
}

public class Poodle extends Dog
 { public Poodle() { }
   public String speak() { return "Vof!"; }
}
```

▶ a)[2] Which of the following subtyping relations are correct?

(i) `Dog<=Animal`, (ii) `Poodle<=Animal`, (iii) `Dog<=Poodle`, (iv) `Animal<=Poodle`

▶ b)[2] Some of the following statements won't be accepted by the compiler. Rewrite them as necessary and briefly mention why they were wrong.

```
Animal x= new Dog();
x.speak();
x.eat();
Dog y= x;
y.eat();
if (x instanceof Dog) { x.eat(); }
```

Now consider the following statements:

```
Dog x= new Dog();
Dog y= new Poodle();
Poodle z= new Poodle();
System.out.println(x.speak());
System.out.println(y.speak());
System.out.println(z.speak());
```

▶ c)[1] What output is generated by these statements?

Exercise 4 (5 points total)

Here is a (somewhat silly) class for Strings:

```
public class SillyString
{ public char first;
  public SillyString rest;

  public SillyString(char c, SillyString S)
  { first= c;
    rest= S;
  }
}
```

For example, the String “Hi” can is created by

```
new SillyString('H', new SillyString('i',null))
```

► a)[1] Create the SillyStrings corresponding to the strings“Gnu” and the empty string “”.

► b)[2] Add a method

```
public int length()
```

to SillyString that returns the number of characters of this SillyString.

► c)[2] Add a method

```
public void removeX()
```

to SillyString that changes this SillyString by removing all occurences of the capital letter ‘X’.