Tentamen i EDAF25

24 augusti, 2023

Skrivtid: 8-13

Lösningsförslag

Grafteori

Uppgift 1

```
 1 markera alla noder som ej besökta
 2 pq = en tom prioritetskö
11         för varje båge e från x:
13             newdist = dist + e.distance
15             om w ej är besökt eller newdist < wdist:
```

Uppgift 2

(a) 4

(b) Algoritmen kommer att hitta vägen A–D–F innan den hittar vägen A–C–E–F. Så fort den hittar den första vägen kommer den avslutas och svara med dess längd, som är 4. Dijkstras algoritm fungerar inte när det finns negativa vikter på bågarna.

(c) Bellman-Fords algoritm klarar grafer med negativa vikter och ger kortaste vägen. För att svara på frågan behöver vi inte kunna Bellman-Fords algoritm utan det är ganska lätt att prova alla möjliga vägar (det finns tre stycken) och se vilken som är kortast. Kortaste vägen är A–C–E–F som har längden 3.

Design

Uppgift 3

```
public class SaveMenuItem extends FileMenuItem {
        private String title;

        public SaveMenuItem(Gui gui, String title) {
                super(gui,title);
        }

        protected int dialogKind(){
                return FileDialog.SAVE
        }

        protected void action(Gui gui, string fullName){
                gui.save(new BufferedWriter(new FileReader(fullName)));
        }
}
```

```
public class OpenMenuItem extends FileMenuItem {
        private String title;

        public OpenMenuItem(Gui gui, String title) {
                super(gui,title);
        }

        protected int dialogKind(){
                return FileDialog.LOAD
        }

        protected void action(Gui gui, string fullName){
                gui.load(new BufferedReader(new FileReader(fullName)));
        }
}

public abstract class FileMenuItem extends MenuItem implements ActionListener {
        private final Gui gui;
        private String title;

        protected FileMenuItem(Gui gui, String title) {
                super(title);
                this.gui = gui;
                this.title = title;
                addActionListener(this);
        }

        protected abstract int dialogKind();
        protected abstract void action(Gui gui, string fullName);

        public void actionPerformed(ActionEvent event) {
                try {
                        FileDialog fileDialog =
                              new FileDialog(gui, title, dialogKind());
                        fileDialog.setVisible(true);
                        String file = fileDialog.getFile();
                        String dir = fileDialog.getDirectory();
                        String fullName = dir + file;
                        fileDialog.dispose();
                        if (file == null) return;
                        action(gui, fullName);
                } catch (Exception ex) {
                        gui.setMessage(ex.toString());
                }
        }
}
```

Uppgift 4

(a)

1. Det är *Dependency Inversion*. Vi vill inte vara beroende av den konkreta klassen PrintStream, istället vill vi bero av ett interface som låter oss skriva ut saker, som sedan kan implementeras av klasser som PrintStream, men även av klasser som vi skriver själva.
2. Vi borde ha något i stil med:

```
interface OutputStream {
    public void println(String s);
}
```

3. Standardklassen PrintStream borde ha deklarationen:

```
class PrintStream implements OutputStream {
    public void println(String s) {
        // ... omissions ...
    }
}
```

(b)

```
interface Command {
    void execute(PC pc, Memory memory, OutputStream output);
}

class PrintCommand implements Command {

    public void execute(PC, Memory memory, OutputStream output) {
        // ... omissions ...
        output.println(???);
    }
}
```

(c)

```
class MemoizingOutputStream implements OutputStream {

    private OutputStream output;
    private String lines = new String();

    public MemoizingOutputStream  (OutputStream output) {
        this.output = output;
    }

    public void println(String s) {
        output.println(s);
        lines = lines+s+"\n";
    }

    public String getLines() {
        return lines;
    }
}
```

(d) För att använda klassen ovan i ett huvudprogram måste vi dels skapa en dekorerad utskriftsenhet runt System.out. Dessutom måste vi skicka den till ett lämpligt ställe. Enklast är att låta run() i Computer ta hand om den, och köra simuleringen mot den aktuella enheten.

Därefter är det enkelt att kontrollera om någon av strängarna i utskriften innehöll texten ”ERROR”:

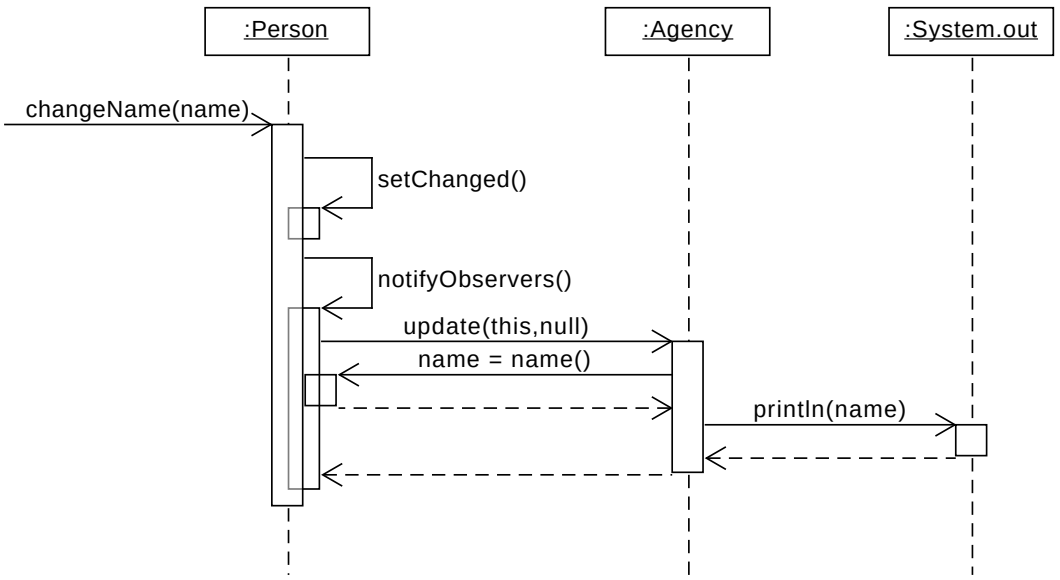
```
void run() {
    var wf = new LongWordFactory();
    var program = new Factorial("5", wf);
    var computer = new Computer(new Memory(1024, wf));
    var output = new MemoizingOutputStream(System.out);
    computer.load(program);
    computer.run(output);
    if (output.getLines().indexOf("ERROR")>=0) {
        System.out.println("Det fanns fel!");
    }
}
```

Uppgift 5

(a)

```
class Person extends Observable {
    private String name;

    public Person  (String name) {
        changeName(name);
    }

    public void changeName(String name) {
        this.name = name;
        setChanged();
        notifyObservers();
    }

    public String name() {
        return name;
    }
}

class Agency implements Observer {
    public void register(Person p) {
        p.addObserver(this);
    }

    public void update(Observable obs, Object obj) {
        var person = (Person) obs;
        System.out.printf("Skatteverket: uppdaterar information om %s\n", person
    }
}
```

(b)



Uppgift 6

Cohesion beskriver hur väl de olika delarna av en mjukvaruenhet (t.ex. en klass) hänger ihop med varandra. Om en enhet innehåller delar som inte har med varandra att göra har den låg cohesion. Om allt i enheten är inbördes relaterat har den hög cohesion. Hög cohesion är bra.

Coupling beskriver hur beroende en mjukvaruenhet är av andra yttre mjukvaruenheter. Det är bra om en enhet är oberoende av andra enheter. Då har den låg coupling.

SRP Single Responsibility Principle. Enligt denna princip bör en mjukvaruenhet ha ett enda ansvarsområde, eller som man också lite (kryptiskt) ibland uttryckt det: Enheten bör bara ”ha ett skäl till att förändras”. Det implicerar också att enheten helst bör ha hela ansvaret för en viss funktionalitet. Därigenom minskar dess coupling samtidigt som det enda ansvarsområdet gör dess cohesion hög.

DIP Dependency Inversion Principle. Principen säger att beroenden bör gå mot abstraktioner snarare än konkreta enheter. Det minskar enhetens coupling genom att den inte beror på en viss specifik yttre mjukvaruenhet.

OCP Open/Closed Principle. Här säger man att en mjukvaruenhet ska vara lätt att utöka med ny funktionalitet utan att man behöver modifiera enheten som sådan. Det kan t.ex. åstadkommas genom generalisering eller genom delegering. Principen bidrar till låg coupling genom att den uppmuntrar till beroende på abstraktioner.