DUGGA: Objektorienterade applikationer

Läs detta!

- *Uppgifterna är inte avsiktligt ordnade efter svårighetsgrad.*
- **Skriv rent dina svar. Oläsliga svar r ä t t a s e j!**
- Programmen skall skrivas i Java 5, eller senare version, vara indenterade och renskrivna och i övrigt vara utformade enligt de principer som lärts ut i kursen.
- Onödigt komplicerade lösningar ger poängavdrag.
- Givna deklarationer, parameterlistor, etc. får ej ändras, såvida inte annat sägs i uppgiften.
- Läs igenom tesen och förbered ev. frågor.

I en uppgift som består av flera delar får du använda dig av funktioner klasser etc. från tidigare deluppgifter, även om du inte löst dessa.

Lycka till!

Uppgift 1 Modellering

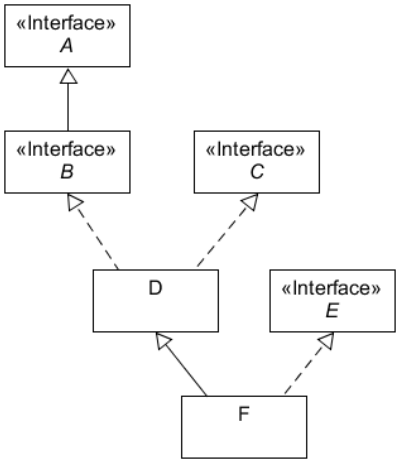
a) Rita ett UML-diagram som beskriver designmönstret Decorator.

(1 p)

b) Rita ett UML-diagram som beskriver följande objektrelationer: En A har en B. B använder C. Många D är associerade till B. En C har en eller flera E.

(1 p)

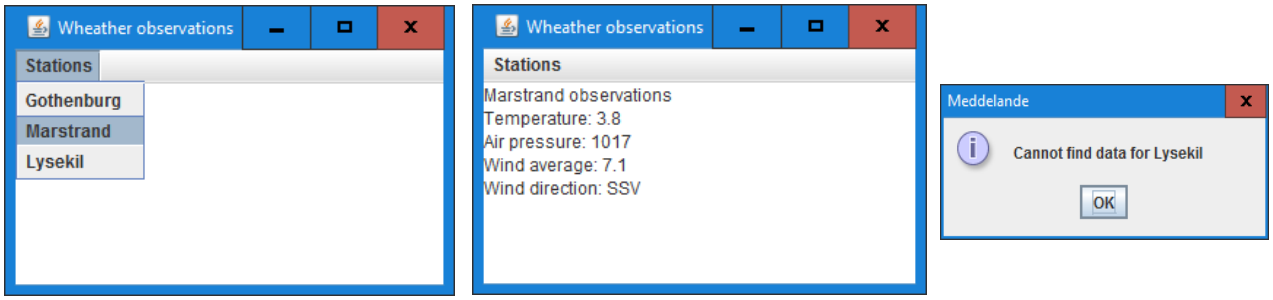
c) Skriv minimala klassdeklarationer (utan kroppar) i Java för klasserna A-F som motsvarar följande UML-diagram.



(1 p)

Grafiska gränssnitt och MVC-modellen

En väderapplikation visar observationsdata för olika platser. Man kan välja plats i en meny och får då diverse uppgifter presenterade i fönstret. Om data saknas för en station får man ett pop-up-meddelande. Följande skärmbilder visar hur det kan se ut:



I de följande två uppgifterna skall du implementera applikationen, dels en filhanteringsdel som laddar väderdata från textfiler, dels ett grafiskt gränssnitt. Applikation skall struktureras enligt designmönstret MVC. I båda uppgifterna används javagränssnittet:

```
public interface DataLoader {
    void loadData(String source) throws IOException;
    String getData();
}
```

Metoden `loadData` läser in väderdata för angiven väderstation, t.ex. `"Gothenburg"`. Du kan anta att filerna med väderdata namnges som *ortnamn*`.txt`. Ex. data för Marstrand ligger i filen `Marstrand.txt`.

Uppgift 2

Konstruera klassen `WheatherDataLoader`. Klassen skall implementera gränssnittet ovan och vara observerbar enligt designmönstret Observer. Metoden `loadData` skall läsa väderdata för angiven station och placera den i en instansvariabel. Texten är lagrad radvis som exemplet i fönstret ovan visar. När du lagrar den skall den ha radstrukturen bevarad. Lägg till `"\n"` där det behövs. Metoden `getData` skall returnera texten om någon lästs in, annars en tom sträng.

(4 p)

Uppgift 3

a) Konstruera klassen `WheatherGui`. Klassen skall vara en observatör enligt designmönstret Observer. Fönstret skall ha en meny enligt exemplet ovan, samt en textarea med 10 rader och 30 kolumner där väderdata kan visas. Textarean skall inte vara redigerbar för användaren. Eventuella filundantag vid användning av `WheatherDataLoader` skall ge en pop-up-dialog som ovan. Dela upp koden i flera metoder.

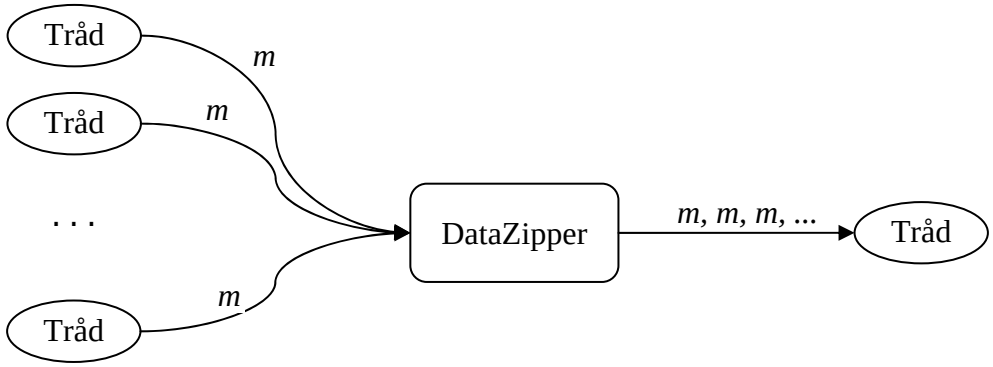
(4 p)

b) Konstruera klassen `Main`. Klassen skall en `main`-metod som skapar och kopplar ihop objekt av klasserna `WheatherDataLoader` och `WheatherGui` till ett fungerande program.

(1 p)

Uppgift 4 Aktiva objekt och trådar

Antag att ett antal trådar producerar meddelanden och att vi vill hantera meddelandena som om de kom från en enda källa. Problemet kan lösas genom att ansluta trådarna till en `DataZipper` som kan ta emot och lagra ett meddelande i taget (metoden `put`) från vilken som helst av producenttrådarna. När ett meddelande tagits emot kan det hämtas av en konsumenttråd med metoden `take`. När detta sker kan `DataZipper`-objektet ta emot ett meddelande igen, o.s.v. i all oändlighet. I vilken ordning meddelandena kommer ut beror på i vilken ordning schemaläggaren låter producenttrådarna lämna meddelanden till `DataZipper`.



a) Konstruera klassen

```
public class DataZipper {
    ...
    public void put(String message) { ... }
    public String take() { ... }
}
```

Inför lämpliga instansvariabler. Observera att det endast skall finnas plats för ett meddelande. Utnyttja javas möjligheter till metodsynkronisering.

(4 p)

b) Konstruera klassen

```
public class Writer {
    public Writer(int id,DataZipper zip) { ... }
}
```

Varje objekt av klassen skall exekvera i sin egen tråd. Var tionde millisekund skall ett meddelande på formen "`(Id=`*i*`,message=`*m*`)`" placeras i `zip`. För varje meddelande skall *m* ökas med ett. Ex. `(Id=7,message=0)`, `(Id=7,message=1)`, …

(3 p)

c) Konstruera klassen

```
public class Reader {
    public Reader(DataZipper zip) { ... }
}
```

Varje objekt av klassen skall exekvera i sin egen tråd och oupphörligt hämta meddelanden från `zip` och skriva ut dem på skärmen.

(2 p)

d) Skriv en `main`-metod som skapar och kopplar ihop *n* `Writer`-trådar med en `DataZipper` och en `Reader`-tråd. Antalet producenter, *n*, skall kunna anges som argument till `main` när programmet körs.

(1 p)

Uppgift 5 Kommunikation

Skriv ett program som oupphörligt läser textrader från tangentbordet och sänder varje rad som ett datagram till en mottagardator vars adress och portnummer ges som programargument vid exekveringen. Eventuella undantag skall leda till lämpliga utskrifter. Man skall alltså kunna exekvera programmet med kommandot

```
> java DatagramSender 123.4.5.6 9876

public class DatagramSender {
    public static void main(String[] arg) {
        ...
    }
}
```

(4 p)

Uppgift 6 Strömmar och filer

Skriv ett program som läser en binärfil med tal av typen **`float`**, samt beräknar och skriver ut medelvärdet av de positiva talen. Om inget medelvärde kan beräknas skall ett meddelande skrivas ut om det. Eventuella undantag skall leda till lämpliga utskrifter. Filnamnet skall ges som programargument:

```
> java AverageComputer numbers.dat

public class AverageComputer {
    public static void main(String[] arg) {
        ...
    }
}
```

(4 p)