NORGES TEKNISK-NATURVITENSKAPELIGE UNIVERSITET
INSTITUTT FOR DATATEKNIKK OG INFORMASJONSVITENSKAP

Faglig kontakt under eksamen:
Dag Svanæs, Tlf: 73 59 18 42

EKSAMEN I FAG
TDT4180 MMI

Torsdag 27. mai 2010
Tid: kl. 0900-1300

Bokmål

Sensuren faller 17. juni 2010

Hjelpemiddelkode: **D** Ingen trykte eller håndskrevne hjelpemidler tillatt.
Bestemt enkel kalkulator tillatt.

Kvalitetssikret: Hallvard Trætteberg

Oppgave 1 (25%) Grensesnittdesign

Læreboka lister opp 8 prinsipper (”golden rules”). Tre av prinsippene er:

- Lag konsistente grensesnitt (”Strive for consistency”)
- Tillat brukeren å angre (”Permit easy reversal of actions”)
- La brukeren ha kontrollen (”Support internal locus of control”)

For hver av disse tre prinsippene, svar på følgende:

- Forklar hva prinsippet går ut på, gjerne med et eksempel.
- Hvorfor er dette er et viktig prinsipp?

Svar:

Lag konsistente grensesnitt (”Strive for consistency”)

- Dette prinsippet handler om at brukergrensesnittet skal se og oppføre seg på samme måte, både internt og i forhold til den omgivende plattformen. Ofte brukes begrepet ”look & feel” for dette, der ”look” handler om det visuelle som for eksempel farger, fonter, layout og valg av knapper etc. ”Feel” handler om interaksjonen, for eksempel knappers oppførsel og navigasjonsstruktur. En annen viktig del av konsistens er bruk av ord og begreper i grensensnittet. Dersom man for eksempel bruker ”Dokument” i en del av et system så er det ikke heldig at det samme begrepet heter ”Fil” i en annen del av systemet.
- Dette er viktig fordi brukeren skal kunne gjenkjenne ord og visuelle uttrykk på en så enkel måte som mulig. Gjenkjennbarhet er viktig for brukervennlighet.

Tillat brukeren å angre (”Permit easy reversal of actions”)

- De fleste IT-systemer tillater at brukeren kan angre på visse handlinger. Det enkleste er ”back” tasten for å kunne angre på å ha tastet feil. Ofte finnes i tillegg en generell ”Angre”/”Undo” som gjør det mulig å angre på handlinger som for eksempel sletting av tekst, forandring av farge. Enkelte ”profesjonelle” systemer, som for eksempel PhotoShop gir brukeren et vindu med hele historikken siden applikasjonen ble startet. Det gjør det mulig å angre seg langt tilbake, og til og med angre på angringen.
- Dette er viktig fordi vi som brukere veldig ofte gjør feil, og ønsker å forandre på noe vi har gjort. Det gjelder spesielt editorer av ymse slag, som for eksempel Word, PhotoShop, Excel, og PowerPoint. Et godt utbygd ”undo” system gir brukeren trygghet og tillater en mer eksperimenterende måte å jobbe på.

La brukeren ha kontrollen (”Support internal locus of control”)

- Dette handler om at datamaskinen ikke skal ta over kontrollen uten at brukeren har bedt om det. Eksempler på det motsatte er Microsoft sin binders (”help clip”) som hele tiden spør brukeren om hun trenger hjelp.

- All erfaring tilsier at brukeren ønsker å ha kontrollen selv i dialog med datamaskiner. Vi ønsker å kunne betrakte de som verktøy som vi bruker for å utføre en jobb, ikke samtalepartnere som man må forholde seg til.

Oppgave 2 (35%) Designprosessen

Standarden ISO 9241-11 definerer brukervennlighet/brukskvalitet (usability) som "*anvendbarhet, effektivitet og subjektiv tilfredsstillelse for spesifikke brukere, med spesifikke mål, i spesifikke omgivelser*".

a) Hvordan måler man typisk hver av de tre faktorene anvendbarhet, effektivitet og subjektiv tilfredsstillelse i en brukbarhetstest?
b) Standarden angir at de tre faktorene måles for "*spesifikke brukere, med spesifikke mål, i spesifikke omgivelser".* Hvilke konsekvenser har dette for planleggingen og utførelsen av en brukbarhetstest?

Svar:

a)

- Anvendbarhet kan enklest måles som antall oppgaver gjennomført uten hjelp. Dette gir et kvantitativt mål på i hvilken grad brukerne faktisk får til å bruke systemet til de oppgavene det er tiltenkt. Eksempel på resultat: *"Gjennomsnittlig gjennomføringsgrad var 85%. Brukerne hadde størst problemer med utskrift og endring av profil."*
- Effektivitet handler om hvor mye ressurser/tid som brukes. Dette kan måles kvantitativt som tid brukt pr. oppgave. Eksempel på resultat: *"Gjennomsnittlig tid brukt på oppgave 1 (pålogging): 45 sekunder. Tre av de 10 testpersonene brukte mer enn 5 minutter på denne oppgaven."*
- Subjektiv tilfredsstillelse kan måles både kvantitativt og kvalitativt. SUS skjemaet er eksempel på et kvantitativt mål, der brukerne fyller ut et skjema om systemet, og det beregnes en score mellom 0 og 100. Eksempel på resultat: *"Gjennomsnittlig SUS score på systemet var 75. Tilsvarende score på systemet de bruker i dag er 70. Vi ser m.a.o. en svak forbedring av opplevd brukervennlighet i forhold til dagens system"*.
- Kvalitativt kan subjektiv tilfredsstillelse måles ved å intervjue testpersonene etter testen. Mye kan også tolkes ut i fra hva de sier under testen.

b)

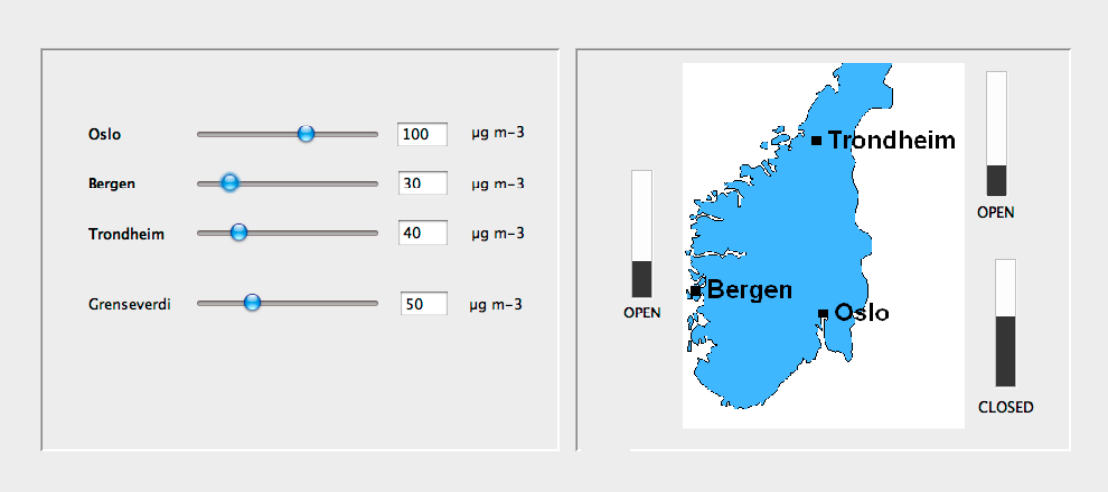
- Standarden sier at brukbarhet må måles for spesifikke brukere med spesifikke mål i spesifikke omgivelser. I forhold til planleggingen av testen så betyr dette at man må finne representative brukere, man må lage representative oppgaver som skal løses, og man må skape realistiske omgivelser. Dette krever kunnskap om brukere, hva de gjør og deres omgivelser. Slik kunnskap kan man få gjennom bl.a. feltstudier, intervjuer, fokusgrupper.
- Under selve testen så er det viktig at man gjennomfører testene slik at de ligner mest mulig på slik det vil være når det ferdige systemet skal brukes. En måte å verifisere realismen på er å spørre brukerne i etterkant om det hele føltes

realistisk. ”Er det slik dere jobber?”. ”Er arbeidsomgivelsene omtrent slik?”. ”Er det noe du kommer på som er annerledes i virkeligheten?”. ”Tror du det som skjedde her er omtrent slik det vil bli når dette systemet kommer på din arbeidsplass?”. ”Føler du at du er en gjennomsnittlig bruker på din arbeidsplass?”.

Oppgave 3 (40%) Grensesnittkonstruksjon

I 2010 har europeisk luftfart blitt forstyrret av aske fra vulkanutbruddet på Island. I den forbindelse så ønsker avisen Morgenposten å tilby sine lesere en daglig oversikt over tilstanden på landets tre viktigste flyplasser i forhold til askeinnhold i luften. Dette vil de gjøre ved å lage et program i Java/Swing der journalisten kan oppdatere måledata og få fram en grafisk kartframstilling som de så kan klippe og lime inn i avisen. Det å klippe og lime er ikke en del av denne oppgaven.

På figuren under ser vi en skisse av hvordan programmet skal se ut.



I ruten til venstre setter journalisten inn askedata og gjeldende grenseverdi. Bildet til høyre viser så automatisk fram søyler for hver by, og teksten ”OPEN” eller ”CLOSED” avhengig om verdien for denne byen er under eller over grenseverdien.

De statiske tekstene (”Oslo”,”Bergen”, ”OPEN”,”CLOSED”, etc.) er realistert v.h.a. *JLabel*. Input-feltene for tall er realisert v.h.a. *JTextField*. Sliderene er realisert v.h.a *JSlider*. Brukeren kan endre askeinnholdet for f.eks. Oslo enten ved å taste inn et nytt tall eller ved å dra i slideren. I begge tilfeller så oppdateres alle relevante deler av grensesnittet automatisk.

Klassen *JTextField* behandler i utgangspunktet ikke tall, men konvertering mellom *String* og *int* kan gjøres enkelt i Java v.h.a innebygde metoder i klassen *Integer.* (Du trenger ikke forholde deg til feilsituasjoner ved at brukeren skriver inn tekst som ikke er gyldige heltall)

static String toString(int i)
Returns a String object representing the specified integer.

Eksempel: s = Integer.toString(i);

static int parseInt(String s)
Parses the string argument as a signed decimal integer.

Eksempel: i = Integer.parseInt(s);

For hver av komponentklassene så er relevante metoder og hjelpeklasser listet i appendikset bakerst. **Merk: Det forlanges ikke at du skal anvende metoder eller klasser som ikke er listet i appendikset.**

a) Hvordan vil du bruke Model-View-Controller (MVC) til å skille presentasjon og data i din løsning av dette programmet? Forklar hovedtrekkene i din løsning i forhold til valg av modell/modeller.

Svar:
Den mest ”rett-fram” løsningen er å la alle data befinne seg i en modell som for eksempel kan hete AshModel. D.v.s. askenivået for hver av de tre byene og grenseverdien. Man kan så ha to view-controllere: En til venstre for sliderne og en til høyre for søylene og tekstene.

En annen løsning er å ha fire separate modeller, en for hver by og en for grenseverdien. De fire modellene kan være instanser av samme modellklasse, som for eksempel kan hete SingleAshModel. Dette vil fungere som klassisk MVC for både slidere og søyler, men tekstene (JLabel) som skal skifte automatisk mellom ”CLOSED” og ”OPEN” vil da måtte ha to modeller, både den byen det gjelder og grenseverdien. Vi har ikke behandlet multiple modeller på samme view i pensum, men det vil fungere fint dersom man sender over riktig ”propertyName” fra modellene ved endring av verdi og har to modellreferanser i tekstviewene.

Vi vil i det følgende ikke utdype løsningen med flere modeller, kun den første med en modell og to view-controllere.

b) Beskriv modellen/modellene i detalj.

Svar:
Modellen har fire tallverdier: Oslo, Bergen, Trondheim og Grenseverdi av type int. Hver av disse har to metoder set… og get…, for eksempel setOslo(int i) og int GetOslo(). I tillegg har hver by en metode isOpen…, for eksempel bool isOpenOslo(), som angir om denne flyplassen er åpen. Det beregnes utifra om askeverdien for denne byen er større eller mindre enn grenseverdien.

Løsningen over gir veldig mange metoder, og skalerer ikke i forhold til å legge til flere byer. Et alternativ er å parametrisere kallet med byene navn, slik at man kan ha vilkårlig antall byer. Vi vil da få ”setCity(String cityName, int Value)”, ”int getCity(String cityName)” og ”bool isOpenCity(String cityName)”. I tillegg: ”setGrenseverdi(int grenseverdi)” og ”int getGrenseverdi()”. Dette vil kreve en tabell eller en annen mekanisme i modellen som gjør at kan holder styr på hvilke byer som har hvilke verdier.

Vi vil i det videre forholde oss til den første løsningen, da den er enklest. Modellen vil måtte ha en instans av klassen PropertyChangeSupport, som holder styr på hvilke view som har denne modellen som sin modell. Metoden addPropertyChangeListener() i modellen sender dette kallet videre til sitt PropertyChangeSupport-objekt. Alle

endringer av verdier, d.v.s. alle ”set” metoder avsluttes med et kall firePropertyChange() til modellens PropertyChangeSupport-objekt. Dette sikrer at det blir sendt endringshendelser (propertyChange) til alle view som har denne modellen som sin modell.

c) Tegn opp hvilke instanser/objekter som eksisterer når programmet kjører. Hva er deres relasjoner/referanser til hverandre? Bruk enkle firkanter for å angi instanser/objekter og piler for å angi relasjoner/referanser.

Svar:

Jeg definerer følgende nye klasser:

- AshApplication inherits JPanel. *Dette er hovedvinduet som holder LeftPanel og RightPanel. Det holder også modellen.*
- AshModel (arver ikke fra noe). *Dette er modellen beskrevet i a) og b).*
- LeftPanel inherits JPanel implements PropertyChangeListener, ActionListener, ChangeListener. *Dette er view-controller for de 4 slidere med tallverdier. Den implementerer interface PropertyChangeListener for å kunne motta endringshendelser fra modellen, ActionListener for å kunne motta brukerhendelser fra tekstfeltene, og ChangeListener for å kunne motta brukerhendelser fra sliderne.*
- RightPanel inherits JPanel implements PropertyChangeListener: *Dette er view-controller for de 3 søylene og CLOSED/OPEN labels. Den implementerer interface PropertyChangeListener for å kunne motta endringshendelser fra modellen.*

Man kunne også ha tenkt seg å pakke inn en slider og tilhørende tallverdi i en egen klasse, men det kompliserer ting litt og jeg velger den enkleste løsningen her.

Følgende instanser finnes:

- Det finnes kun én AshModell-instans. Den har et felt pcs som peker på en instans av klassen PropertyChangeSupport som brukes til å administrere lyttere.
- Det finnes en instans av hver LeftPanel og RightPanel. Disse har begge et felt ”model” som peker på modellen. Metoden setModel() brukes til å sette modellen.
- En instans av AshApplication holder AshModel-modellen og LeftPanel- og RightPanel-instansene.
- LeftPanel-intansen har 4 slidere og 4 tekstfelt. Disse har hvert sitt felt slik at man kan kjenne igjen hvor brukerhendelsene kommer fra ved å plukke ut kilden i eventene v.h.a. getSource().
- RightPanel-instansen har 3 søyler og tre labels for ”CLOSED/OPEN”.
- Søylene kan implementeres v.h.a. en sort JPanel i en hvit JPanel. Høyden på den sorte JPanel kan så endres avhengig av verdien. Man kunne også ha laget en egen klasse Bar som tar seg av dette. Den ville da ha metoden setValue() på samme måte som en slider.

d) Tegn opp sekvensdiagrammet når det hele startes opp. Forklar.

Svar:

Jeg antar at alle instanser er skapt.

- Modellen må kobles til de to panelene. Oppstart av dette skjer fra sist i AshApplication-instansen sin oppstartskode. Det skjer v.h.a. kallene setModel() til panelene. Fra setModel() i panelene så kalles modellen med addPropertyChangeListener() med seg selv som parameter.
- Så må brukerhendelsene havne på riktig sted. Det skjer fra sist i LeftPanel-instansen sin oppstartskode. For hver av de 4 sliderne kalles addChangeListener() med panelet selv som parameter. For hver av de 4 tekstfeltene kalles addActionListener med panelet selv som parameter.
- RightPanel har ingen komponenter som kan motta brukerhendelser, og vi trenger følgelig ingen event-lyttere her.

e) Tegn opp sekvensdiagrammet når brukeren legger inn et nytt tall i tekstfeltet for ”Grenseverdi”. Forklar.

Svar:

1. Et nytt tall i tekstfeltet for grenseverdi vil skape et actionPerformed()-kall fra denne JTextField-instansen tilbake til LeftPanel-instansen med et ActionEvent-objekt som parameter.
2. LeftPanel-instansen kaller så Action-Event objektet med getSource() for å finne ut hvilket tekstfelt som genererte brukerhendelsen.
3. LeftPanel-instansen finner ut at det var grenseverdien som ble endret, og kaller modellen med setGrenseverdi() med den nye grenseverdien som parameter. Denne er først konvertert fra tekst til tall.
4. I modellen så settes den riktige verdien i grenseverdi-feltet, og den kaller så sitt PropertyChangeSupport objekt med firePropertyChange() med ”Grenseverdi” som parameter.
5. Dette genererer to propertyChange()-kall til de objektene som har denne modellen som sin modell, ett til LeftPanel- og ett til RightPanel-instansen.
6. propertyChange()-kallet til LeftPanel-instansen fører først til et getPropertyName-kall til PropertyChangeEvent objektet for å avgjøre hva som er endret. Verdien ”Grenseverdi” tolkes til at det er slideren og tekstfeltet for grenseverdi som skal endres.
7. LeftPanel-instansen kaller så modellen med getGrenseverdi() for å få ny verdi for grenseverdi.
8. LeftPanel-instansen kaller så grenseverdi-slideren med setValue() med denne verdien.
9. LeftPanel-instansen konverterer så tallverdien til en tekst og kaller tekstfeltet med setText().
10. propertyChange()-kallet til RightPanel-instansen fører først til et getPropertyName-kall til PropertyChangeEvent objektet for å avgjøre hva som er endret. Verdien ”Grenseverdi” tolkes til at alle de tre JLabel potensielt må endres.
11. RightPanel-instansen kaller så modellen med isOpenOslo() for å finne ut om den tilsvarende JLabel skal være ”CLOSED” eller ”OPEN”.
12. JLabel for Oslo settes så til for eksempel ”CLOSED” v.h.a. kallet setText() med ”CLOSED” som parameter.

13. Tilsvarende kalles modellen med isOpenBergen() og isOpenTrondheim(), og de tilhørende JLabels settes til riktig verdi.

Du trenger ikke forholde deg til layout, figuren av kartet eller hvordan denne tegnes ut.

Hint: Et JPanel eller en subklasse av JPanel kan godt være usynlig (ha samme farge og rammefarge som bakgrunnen) og brukes til å samle flere Swing-komponenter.

Appendiks: Relevante klasser, interface og tilhørende metoder

Det følgende er klippet og limt fra den offisielle dokumentasjonen på java.sun.com.

class PropertyChangeSupport

This is a utility class that can be used by objects that support bound properties. You can use an instance of this class as a variable in your object and delegate various work to it.

Methods:

public void addPropertyChangeListener(PropertyChangeListener listener)

Add a PropertyChangeListener to the listener list. The listener is registered for all properties.

Parameters:
listener - The PropertyChangeListener to be added

public void firePropertyChange(String propertyName,
Object oldValue,
Object newValue)

Report a bound property update to any registered listeners. No event is fired if old and new are equal and non-null.

Parameters:
propertyName - The name of the property that was changed.
oldValue - The old value of the property.
newValue - The new value of the property.

interface PropertyChangeListener

A "PropertyChange" event gets fired whenever an object changes a "bound" property. You can register a PropertyChangeListener with a source object so as to be notified of any bound property updates.

Methods:

public void propertyChange(PropertyChangeEvent evt)

This method gets called when a bound property is changed.

Parameters:
evt - A PropertyChangeEvent object describing the event source and the property that has changed.

class PropertyChangeEvent

A "PropertyChange" event gets delivered whenever an object changes a "bound" or "constrained" property. A PropertyChangeEvent object is sent as an argument to the PropertyChangeListener method.

Normally PropertyChangeEvents are accompanied by the name and the old and new value of the changed property. Null values may be provided for the old and the new values if their true values are not known.

An event source may send a null object as the name to indicate that an arbitrary set of if its properties have changed. In this case the old and new values should also be null.

Methods:

public String getPropertyName()

Gets the programmatic name of the property that was changed.

Returns:
The name of the property that was changed. May be null.

public Object getNewValue()

Gets the new value for the property, expressed as an Object.

Returns:
The new value for the property. May be null.

public Object getOldValue()

Gets the old value for the property, expressed as an Object.

Returns:
The old value for the property. May be null.

class JPanel

JPanel is a generic lightweight container.

Methods:

public Component add(Component comp)

Appends the specified component to the end of this container.

class JTextField

JTextField is a lightweight component that allows the editing of a single line of text.

Methods:

public String getText()

Returns the text contained in this JTextField

public void setText(String t)

Sets the text of this JTextField to the specified text.

public void addActionListener(ActionListener l)

Adds the specified action listener to receive action events from this JTextField.

interface ActionListener

The listener interface for receiving action events. The class that is interested in processing an action event implements this interface, and the object created with that class is registered with a component, using the component's addActionListener method. When the action event occurs, that object's actionPerformed method is invoked.

Methods:

public void actionPerformed(ActionEvent e)

Invoked when an action occurs.

Class ActionEvent

A semantic event which indicates that a component-defined action occured. This high-level event is generated by a component (such as a Button) when the component-specific action occurs (such as being pressed). The event is passed to every ActionListener object that registered to receive such events using the component's addActionListener method.

The object that implements the ActionListener interface gets this ActionEvent when the event occurs. The listener is therefore spared the details of processing individual mouse movements and mouse clicks, and can instead process a "meaningful" (semantic) event like "button pressed".

Methods:

public Object getSource()

The object on which the Event initially occurred.

Returns:
The object on which the Event initially occurred.

Class JSlider

A component that lets the user graphically select a value by sliding a knob within a bounded interval.

Methods:

public int getValue()

Returns the sliders value.

public void setValue(int n)

Sets the sliders current value.

public void addChangeListener(ChangeListener l)

Adds a ChangeListener to the slider.

Interface ChangeListener

Defines an object which listens for ChangeEvents.

public void stateChanged(ChangeEvent e)

Invoked when the target of the listener has changed its state.

Class ChangeEvent

ChangeEvent is used to notify interested parties that state has changed in the event source.

Methods:

public Object getSource()

The object on which the Event initially occurred.

Returns:
The object on which the Event initially occurred.

class JLabel

A display area for a short text string. A label does not react to input events.

Methods:

public void setText(String text)

Defines the single line of text this component will display.