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

Onsdag 18. mai 2009
Tid: kl. 0900-1300

Bokmål

Sensuren faller 8. juni 2008

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

Kvalitetssikret: Hallvard Trætteberg

Oppgave 1 (25%) Grensesnittdesign

a) Forklar begrepet *mental modell*, og vis med et eksempel hvordan dette begrepet er relevant for utformingen av brukergrensesnitt.

En mental modell er brukerens modell av virkemåten og strukturen til et produkt. Dette begrepet er relevant fordi det er viktig å vite hvordan brukeren forstår strukturen og virkemåten til det vi lager.

Eksempel: En termostat.
Brukeren må forstå den grunnleggende virkemåten, ellers så vil vi få adferd som er uønsket. En termostat kan forstås av brukeren på flere måter. Den korrekte måten, d.v.s. den som passer med designerens konseptuelle modell, er at man setter ønsket temperatur og så slår varmekilden av eller på avhengig av om målt temperatur er over eller under ønsket temperatur. Forenklet så kan man si at man setter ønsket temperatur og så blir det den temperaturen. En alternativ mental modell er at man vrir på en knapp, og jo høyere man vrir den jo mer varme får man, altså som på en komfyr. Dersom er bruker har feil mental modell så kan det for eksempel føre til at man kommer hjem i et kaldt hus og setter termostaten på 40 grader fordi man ønsker å få det varmt så fort som mulig. Hvis man da drar bort en halv dag så kommer man hjem til 40 grader, og ikke 20 som man egentlig ønsket. Feil mental modell kan føre til uønsket adferd.
Den ”riktige” mentale modellen kalles for konseptuell modell. I utformingen av brukergrensesnitt er det viktig å formidle den konseptuelle modellen på en slik måte at brukeren får den riktige mentale modell.

b) Forklar begrepet *affordance*, og vis med et eksempel hvordan dette begrepet er relevant for utformingen av brukergrensesnitt.

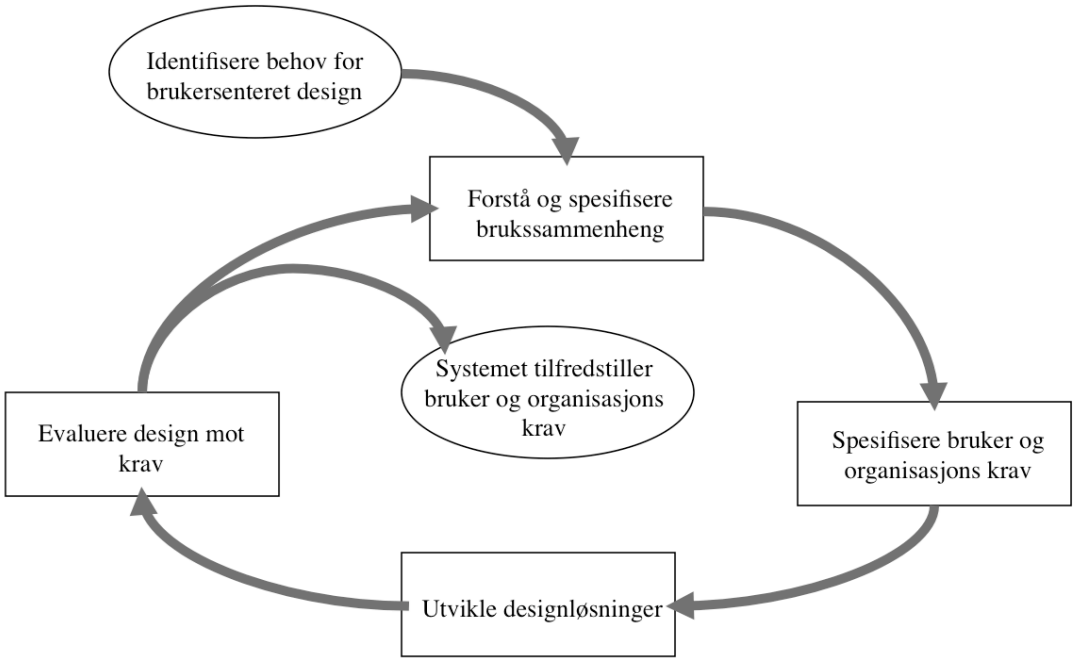
Affordance er den bruk et objekt signaliserer gjennom sin form og farge. Det finnes forskjellig typer affordance: gitt av menneskekroppen, kulturelt, og kontekstuelt. Relevansen for utformingen av brukergrensesnitt er at det er viktig å signalisere hva som kan trykkes på i et grafisk grensesnitt. Spesielt er dette viktig når man ikke har en 3D form som for eksempel i produktdesign (for eksempel Cola-flasken). Med en skjermflate så må man skille ”forgrunn” (det som kan trykkes på) fra ”bakgrunn” (det som ikke kan trykkes på) v.h.a. form, farge og tekst. Problemer med manglende ”affordance” er veldig vanlige i skjermdesign, og er en av grunnene til at det er viktig å bli enige om standardelementer i GUI.

Eksempel: Dersom man har et bilde av en kamera i et skjermbilde. Er det trykkbart? Hva fører det i tilfelle til dersom man trykker på det? Eller er det bare pynt i bakgrunnen? Dersom man for eksempel ønsker at dette skal være en knapp som kan trykkes på for å ta vare på skjermbildet til fil, så må man indikere det på en eller annen måte. På alle GUI plattformer så er det innført konvensjoner på hvordan bl.a. knapper ser ut. Ved å gjøre kamerabildet om til et knappeikon så gjør man bruk av en tillært, kulturell affordance. For også å forklare hva som skal skje når man trykker på knappen så kan man sette på en tekst, for eksempel ”Save screen”.

Oppgave 2 (35%) Designprosessen

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

ISO 13407 beskriver en brukersentrert prosess bestående av fire faser i en sirkelbevegelse. Figuren under er hentet fra standarden og illustrerer dette.



a) Gi ett eksempel på en aktivitet/teknikk (noe man gjør) for hver av de fire fasene i figuren over.

Det blir her bedt om kun en metode pr. fase. Mange forskjellige metoder kunne ha vært anvendt, men jeg tar her kun en typisk metode pr. fase. Et prosjekt vil, som ISO 13407 viser, gå igjennom fasene flere ganger. Man endrer da ofte metode tilpasset hvor langt man er kommet i prosjektet. Jeg angir her kun en typisk første iterasjon.

I. Hensikten her er å forstå brukssammenhengen (context-of-use). Det er da viktig å få innblikk i brukerens hverdag og den sammenheng som produktet skal brukes i. Jeg ville da ha valgt å gjøre et feltstudie. Det innebærer å få tilgang til brukeren, og følge en eller flere brukere over en periode. Man gjør seg da notater, og spør brukerne om det som skjer for å få utdypet sine observasjoner. Ut i fra et feltstudie så får man innblikk i hvem brukerne er, hva de holder på med, og i hvilke omgivelser de befinner seg.

II. Hensikten her er å få på plass krav til produktet. Jeg ville da ha invitert brukere og bestillere til en designworkshop. Hensikten ville ha vært å få fram krav til systemet. Jeg ville da ha begynt med å presentere funnene fra feltstudiet, og min/vår analyse av dette.
III. Ut fra kravene så ville jeg ha gjort en prototyping. Det innebærer å lage en tidlig utgave av det som skal bli produktet. I en tidlig fase så kan man benytte papirprototyper, for så senere å lage interaktive prototyper.
IV. Jeg ville ha evaluert designet gjennom en brukbarhetstesting i lab med potensielle brukere. Det gir feedback som kan danne grunnlag for neste iterasjon i designprosessen.

b) Beskriv hvordan definisjonen av brukervennlighet i ISO 9241-11 kan brukes som rettesnor for hver av de fire aktivitetene du har valgt.

For hver av de 4 fasene så sier ISO 9241-11 at man må kjenne brukerne, deres oppgaver og deres omgivelser. Den sier også noe om hva som kan/skal måles – anvendbarhet, effektivitet og subjektiv tilfredsstillelse.

Feltstudie:
Her er viktigste fokus på brukskontekst. ISO 9241-11 sier at brukervennlighet er i forhold til bestemte brukere med bestemte mål i bestemte omgivelser. Dette er en fin rettesnor når vi gjør våre observasjoner. Hvem er brukerne? Hva holder de på med som er relevant for det produktet som skal lages? I hvilke fysiske og sosiale omgivelser skjer dette? Hvis vi for eksempel skal lage en ny interaktiv informasjonskiosk for NSB, så vil vi typisk dra på feltstudie til en jernbanestasjon for å observere de reisende. Resultatet kan formidles gjennom for eksempel noe typiske personas og deres bruksscenarier.
Feltstudiet kan også avdekke hva som er viktige suksessfaktorer for produktet. Den første inndeling kan være å skille mellom anvendbarhet, effektivitet og subjektiv tilfredsstillelse. I eksempel med infokiosk på jernbanestasjon så vil man for eksempel kunne finne at effektivitet er viktigere enn kult design fordi de reisende har dårlig tid.

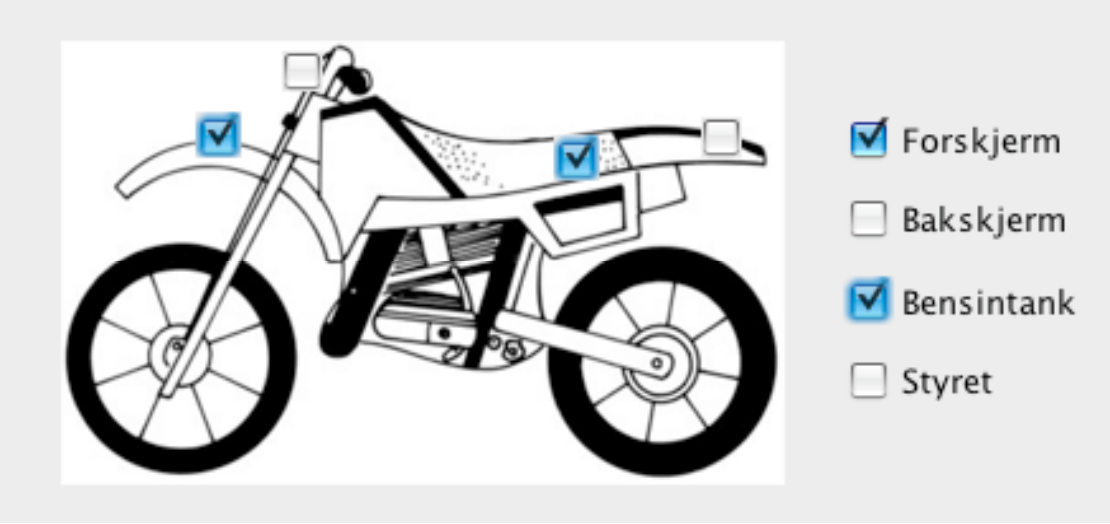
Designworkshop:
Igjen kan standarden brukes som rettesnor, men nå ikke i forhold til datainnsamling men i forhold til hva et kravdokument skal inneholde. Standarden kan fungere som en huskeliste over ting som man må bli enige om: hvem er brukerne (NSB eksempelet: Kun norske reisende? Barn? Funksjonshemmede?), hva skal de kunne gjøre (Se togtidene? Lære om reisemål? Finne steder å spise?,,), hvilken omgivelse (Hvor skal infokiosken stå?). Man må også bli enige om krav til effektivitet og brukertilfredshet. Hvor lett skal det være å bruke infokiosken?

Prototyping:
Her er standarden viktig ved at man forholder seg til kravene fra designworkshopen og dataene fra feltstudiet når man designer. Dette gir kunnskap om hvem brukerne er og hva systemet skal gjøre. Når man designer prototypen så må man hele tiden ha brukerne, bruken og omgivelsene i bakhodet. Ofte så er det designvalg som må gjøres som ikke kom fram av kravdokumentet. Det er da viktig å ha dataene fra tidlige faser å støtte seg på. (Eksempel NSB: Dersom infokiosken skal brukes av barn så må man ikke ha for lange og vanskelige ord).

Brukbarhetstest:
Her er standarden direkte anvendbar for å bygge opp testen. Brukere: Det er viktig å bruke riktige testpersoner som er representative for de mest typiske brukergruppene (NSB: Eldre, Barn, Interrail,,). Oppgaver: Det er viktig å gi brukerne typiske og realistiske oppgaver som skal løses i brukbarhetstesten (NSB: Finnes togtider, Finne sushibar i Trondheim,,). Omgivelser: Det er viktig å gjenskape realistiske omgivelser (NSB: Mye støy, Liten fysisk plass og mange mennesker,,). Dersom det er satt krav til for eksempel effektivitet så må også dette testes for.

Oppgave 3 (40%) Grensesnittkonstruksjon

Et bil- og motorsykkelutleiefirma ønsker å lage et Java/Swing-basert skadeskjema for bil og motorsykkel. Denne oppgaven handler kun om skadeskjema for motorsykkel.



I figuren ser vi et eksempel på et skadeskjema for motorsykkel. Brukeren skal kunne krysse av for skade enten på selve motorsykkelen eller i listen til høyre. Kryssene (JCheckBox) skal i begge tilfeller oppdateres automatisk.

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

Model-View-Controller (MVC) er en software arkitektur som muliggjør å skille datalaget fra presentasjonslaget i en applikasjon. I Java/Swing er View og Controller ofte sydd sammen i ferdige GUI komponenter som for eksempel JButton. Datene/tilstandene legges i modell-objekter. Disse kobles til View-Controller. Når en verdi endres i et View så er det viewets ansvar å oppdatere modellen. Endringer i modellen propageres så til alle Views som har denne modellen som sin modell. På denne måten vil tilstanden til alle Views til enhver tid være oppdatert fordi dataene kun ligger ett sted, nemlig i modellen.

I denne oppgaven ser jeg at de samme dataene gjentas i bildet til venstre og i listen til høyre. De underliggende data er 4 stk. ”ja/nei” variable (boolean). Jeg vil da realisere dette v.h.a. to views som deler samme modell, et view for bildet til venstre og et view for tabellen til høyre.

De to viewene har likt utseende og oppførsel, med unntak av layout på skjermen. Det står i oppgaven at jeg ikke skal forholde meg til selve figuren av motorsykkelen og hvordan denne tegnes ut. Jeg velger derfor å ikke beskrive layout problematikken. Ettersom de to viewene er så like, så vil jeg lage en abstrakt superklasse RapportView

som inneholder alt som er felles, og definere detaljene med layout i to subklasser av denne: VenstreRapportView og HoyreRapportView.

b) Beskriv modellen/modellene i detalj.

Egen modell:

Jeg vil lage en modell RapportModel som ikke arver fra noen annen klasse. Vi bygger på JavaBean måten å bygge opp en modell.

Variable:
4 x boolean for h.h.v. forskjerm, bakskjerm, bensintank og styret. Initielt false.
Disse er private, og er følgelig ikke synlig utenfor objektet.

Accessmetoder:
For hver av de fire variablene vil jeg lage en set og en get metode som er ”public”, for eksempel setForskjerm og getForskjerm. De er da synlige utenfor objektet. Metoden setForskjerm tar en Boolean som argument, mens getForskjerm returnerer en Boolean.

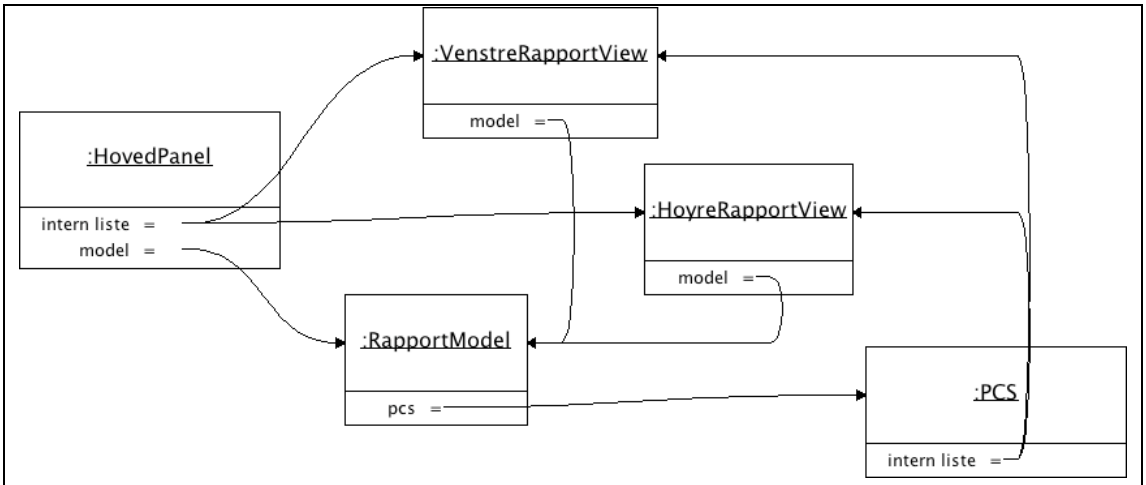
Kode for oppdatering:
Klassen PropertyChangeSupport gir støtte for oppdatering av tilkoblede Views. Jeg vil har et eget PCS-objekt i modellen for å holde på listen over avhengige views.

Metode addPropertyChangeListener for å legge til lyttere.

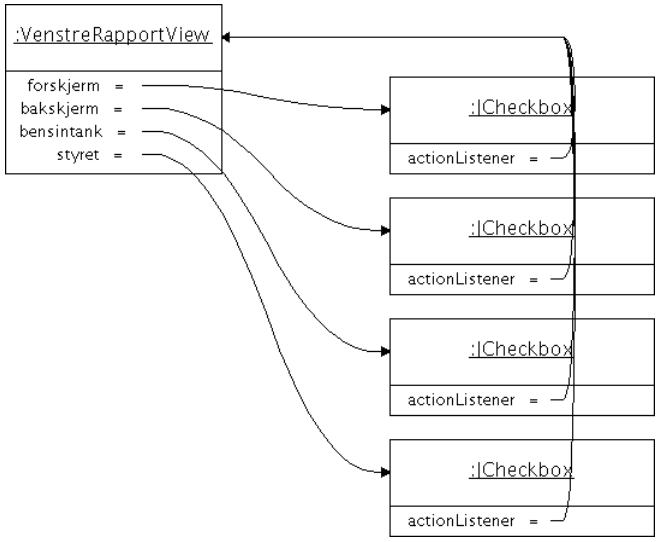
Alle kall til set-metoder fører til at PCS-objektet får meldingen firePropertyChange. Dette fører til at det sendes update meldinger til alle lyttere.

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

Jeg har brutt opp figuren i en hovedfigur og en detaljfigur.



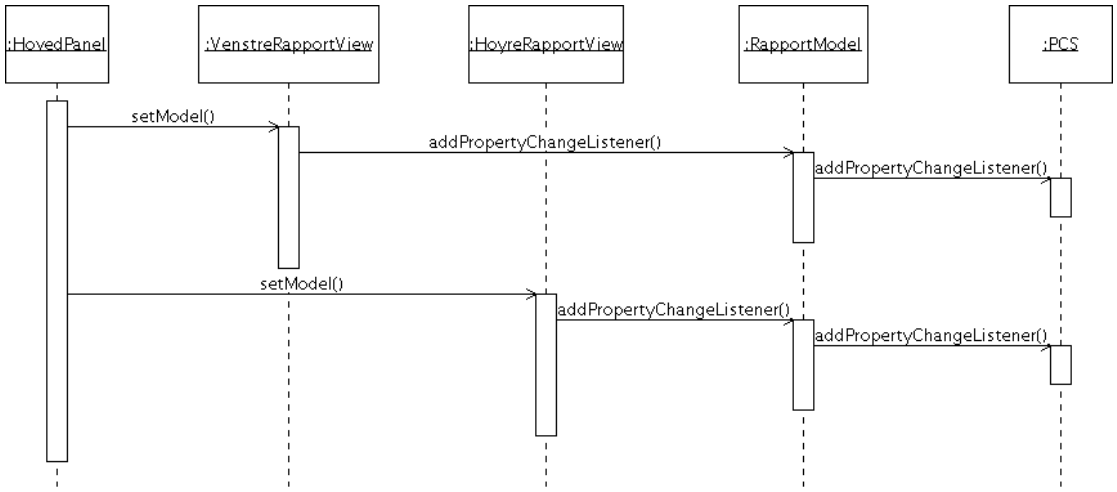
Vi ser over et objektdiagram med hovedstrukturen. Det er et hovedpanel som eier de to viewene og en modell. Veiewene har modellen som modell, ved at deres modellvariable peker på modellen. Internt så har modellen et PCS objekt. Dette har en intern liste som peker på de to viewene.



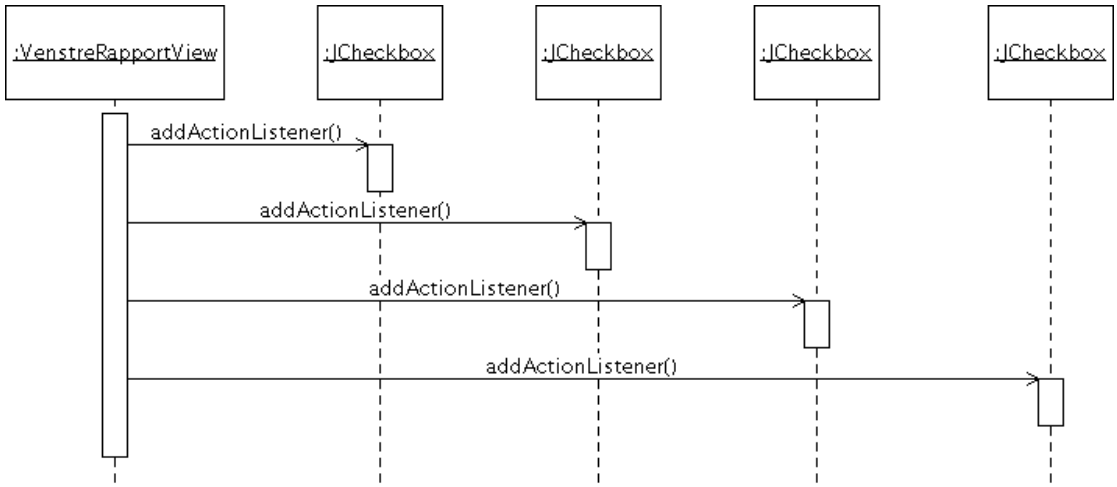
Vi ser over detaljene for et av Viewene. Det er likt for begge. Viewet har fire jCheckbox. Disse har dette viewet som sin actionlistener. Det kan enten gjøres ved at selve viewet er actionlistener, eller ved at det finnes indre klasser i dette viewet som mottar brukerhendelser. Feltet ”actionListener” angir verdien av den interne listen av lyttere for JCheckbox.

d) Tegn opp sekvensdiagrammet når skadeskjemaet startes opp. Forklar.

Jeg har brutt opp dette i to diagrammer.

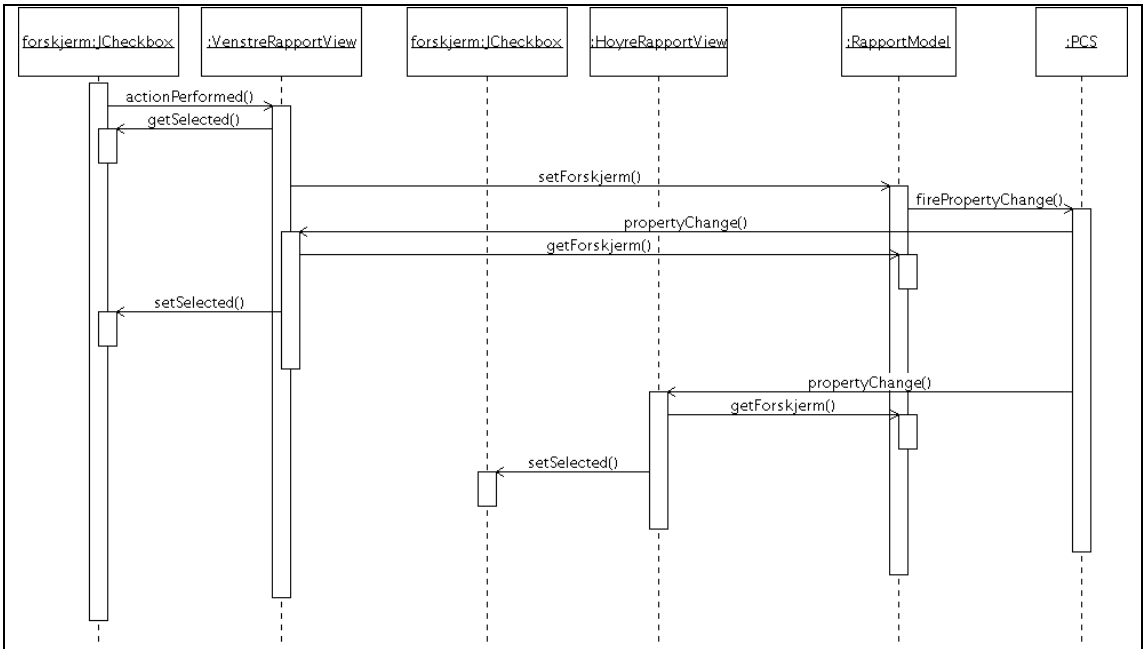


Diagrammet over viser koblingen til modellen. Hovedpanelet kaller hvert av viewene med setModel() med modellen som parameter. Viewet tar vare på referansen i sin model varabel og legger seg selv som lytter på endringshendelser ved å kalle modellen med addPropertyChangeSupport() med seg selv som parameter. Internt i modellen så videresendes denne meldingen til modellens PCS objekt.



Diagrammet over viser oppkoblingen av lyttere fra et view til sine JCheckboxer. Det er her vist som om det er selve viewet som kobles på som lytter. I en faktisk implementasjon så ville jeg ha valgt å lage en indre klasse pr. JCheckbox som mottok hendelser, men det blir litt unødvendig komplisert å vise fram.

e) Tegn opp sekvensdiagrammet når brukeren klikker på forskjermen på figuren for å indikere at det er en skade og den tilsvarende sjekkboksen til høyre oppdateres. Forklar.



Det er her vist kun en JCheckbox pr. view. Det vil være fire pr. view.

Det som skjer:

1. Forskjerm sjekkboksen i venstre view blir valgt ofg sender actionPerformed() til sin eiende view.
2. Viewet trenger å vite verdien på sjekkboksen, og sender forespørselen getSelected() til JCheckBox.
3. Melding setForskjerm(true) til viewet sin modell.
4. Melding firePropertyChange fra modell til PCS.
5. Melding propertyChange fra PCS til første view.
6. View spør modell om getForskjerm
7. View kaller setSelected til forskjerm-JCheckBox.
8. Gjentas for alle JCheckBox og alle (begge) views.

f) Det er ønskelig med et tekstfelt (JLabel) i tillegg som oppsummerer skadene tekstlig. For utfyllingen i eksempelet over ville det da stå: ”Du har rapportert skade på Forskjerm og Bensintank.”. Hvordan ville du implementere et slikt tillegg, og hvilke fordeler gir det at man valgte en MVC-arkitektur for implementasjonen.

I. Jeg ville hatt en JLabel med i et eget view som het MyLabelView (subklasse av JPanel). Koble dette viewet til modellen. Alternativt ville jeg ha laget en adapter som genererte teksten. Et annet alternativ er å la selve eiende view ha modellen som modell. Det siste er noe rotete.

II. Klassen MyLabelView vil, som de andre to viewene, implementere propertyChangeListener. Når den mottar propertyChange så spør den modellen om de 4 booleans og genererer riktig tekst som settes i JLabel.

Du trenger ikke forholde deg til selve figuren av motorsykkelen og hvordan denne tegnes ut.

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

Vedlagt ligger et kodeeksempel. Dette skal **ikke** gjøres på eksamen. Det skal **ikke** være kode på eksamen!!! Det er her kun for å vise hvordan den resulterende koden kunne blitt seende ut. Koden er kun for 2 stk. JCheckbox pr. View. Den har ikke noe spesiell kode for layout. Fokus er på å vise MVC-mekanismene.

```
import javax.swing.JPanel;
import javax.swing.JFrame;

public class ReportMain extends JPanel {
	private ReportView reportView1, reportView2;

	private ReportModel model;

	public ReportMain() {
		model = new ReportModel();
		reportView1 = new LeftReportView();
		reportView1.setModel(model);
		add(reportView1);

		reportView2 = new RightReportView();
		reportView2.setModel(model);
		add(reportView2);
	}

	public static void main(String args[]) {
		JFrame frame = new JFrame("...");
		frame.getContentPane().add(new ReportMain());
		frame.pack();
		frame.setVisible(true);
	}
}
```

```
import java.beans.PropertyChangeSupport;
import java.beans.PropertyChangeListener;

public class ReportModel {
	public final static String FRONT_FENDER_PROPERTY = "Front fender";
	public final static String REAR_FENDER_PROPERTY = "Rear fender";
	private PropertyChangeSupport pcs;
	private Boolean frontFender = false;
	private Boolean rearFender = false;

	public ReportModel() {
		pcs = new PropertyChangeSupport(this);
	}

	public Boolean getFrontFender() {
		return frontFender;
	}

	public void setFrontFender(Boolean newValue) {
		Boolean oldValue = this.frontFender;
		this.frontFender = newValue;
		pcs.firePropertyChange(FRONT_FENDER_PROPERTY, oldValue, newValue);
	}

	public Boolean getRearFender() {
		return rearFender;
	}

	public void setRearFender(Boolean newValue) {
		Boolean oldValue = this.rearFender;
		this.rearFender = newValue;
		pcs.firePropertyChange(REAR_FENDER_PROPERTY, oldValue, newValue);
	}

	public void addPropertyChangeListener(PropertyChangeListener listener) {
		pcs.addPropertyChangeListener(listener);
	}

}
```

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

public abstract class ReportView extends JPanel implements PropertyChangeListener {

        private ReportModel model;
        protected JCheckBox frontFenderBox;
        protected JCheckBox rearFenderBox;

        public ReportView() {
                frontFenderBox = new JCheckBox();
                add(frontFenderBox);
                frontFenderBox.addActionListener(new MyFrontFenderBoxAction());
                rearFenderBox = new JCheckBox();
                add(rearFenderBox);
                rearFenderBox.addActionListener(new MyRearFenderBoxAction());
                subClassLayoutCode();
        }

        abstract void subClassLayoutCode();

        /** Action for clearButton * */
        class MyFrontFenderBoxAction implements ActionListener {
                public void actionPerformed(ActionEvent e) {
                        model.setFrontFender(frontFenderBox.isSelected());
                }
        }

        class MyRearFenderBoxAction implements ActionListener {
                public void actionPerformed(ActionEvent e) {
                        model.setRearFender(rearFenderBox.isSelected());
                }
        }

        // Set model
        public void setModel(ReportModel theModel) {
                model = theModel;
                model.addPropertyChangeListener(this);
        }

        // PropertyChangeListener
        public void propertyChange(PropertyChangeEvent evt) {
                if (evt.getPropertyName() == ReportModel.FRONT_FENDER_PROPERTY) {
                        frontFenderBox.setSelected(model.getFrontFender());
                } else if (evt.getPropertyName() == ReportModel.REAR_FENDER_PROPERTY) {
                        rearFenderBox.setSelected(model.getRearFender());
                }
        }
}
```

```
public class LeftReportView extends ReportView {

         void subClassLayoutCode() {

         };
}
```

```
public class RightReportView extends ReportView {

        void subClassLayoutCode() {
                 frontFenderBox.setText("Front");
                 rearFenderBox.setText("Rear");
        }

}
```

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 JCheckBox

An implementation of a check box -- an item that can be selected or deselected, and which displays its state to the user.

Methods:

public void addActionListener(ActionListener l)

Adds an ActionListener to the button.

Parameters:
l - the ActionListener to be added

public boolean isSelected()

Returns the state of the button. True if the toggle button is selected, false if it's not.

Returns:
true if the toggle button is selected, otherwise false

public void setSelected(boolean b)

Sets the state of the button. Note that this method does not trigger an actionEvent. Call doClick to perform a programatic action change.

Parameters:
b - true if the button should be selected, otherwise false

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.

MERK: Det er ikke nødvendig å benytte alle metoder/funksjoner.