Laboration nr 3
Memory-spel

Syfte

Syftet med denna laboration är att ge erfarenhet av att utveckla ett något större program som använder grafik, händelsestyrning samt är strukturerat enligt designmönstret Model-View-Controller.

Redovisning

Dokumentationen skall förutom källkoden (.java-filer), innehålla ett klassdiagram i UML som beskriver relationerna mellan klasserna (.pdf-dokument).

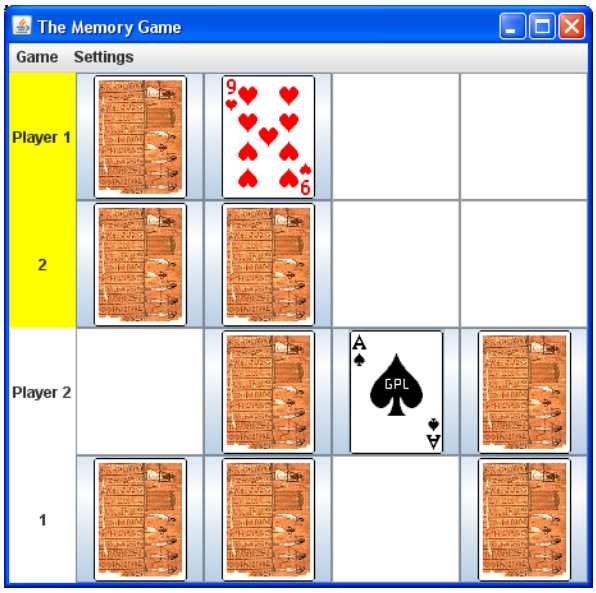
Källkoden skall vara välstrukturerad, använda lämplig beskrivande namngivning och vara snyggt redigerad med tydlig och konsekvent indentering. Klasser och metoder skall dokumenteras med JavaDoc. Bifoga dessutom katalogen med de bildfiler som programmet använder.

Dokumentationen skall skickas in via Fire. Glöm inte att trycka på SUBMIT!

Problembeskrivning

Spelet *Memory* kan spelas av en eller flera spelare. Till spelet hör ett antal kort med samma baksida men med olika bilder på framsidan. Varje bild förekommer på exakt två kort. När spelet börjar läggs korten ut på bordet med baksidan uppåt. Spelarna får sedan i tur och ordning vända på två kort. Om de två korten har olika bilder vänds de tillbaka så att baksidan kommer uppåt. Nästa spelare får i så fall fortsätta. Om de två korten däremot har samma bild får spelaren ta korten från bordet och lägga dem i sin egen hög. Samma spelare får sedan fortsätta att vända två nya kort. En spelare får fortsätta så länge som han eller hon lyckas hitta kortpar med samma bild. Spelet fortsätter tills alla kort på bordet är borta. Den spelare som då har flest par har vunnit.

Uppgiften är att skriva ett javaprogram som låter användarna spela Memory. Istället för att ha kort på bordet skall korten visas på skärmen. Spelarna väljer kort genom att klicka på korten. När en spelare har klickat på sitt andra kort skall korten visas 2 sekunder, varefter de automatiskt vänds så baksidan kommer upp eller tas bort beroende på om korten var ett par eller inte. I programmet skall det finnas ett fönster med ett antal kort. Programmet skall utformas så att antalet rader och kolumner kan bestämmas dynamiskt när man startar programmet. I fönstret skall också finnas en meny med val för att starta ett nytt spel eller avsluta programmet. Programmet skall kunna hantera två spelare. De två spelarnas aktuella poäng, dvs. antalet kortpar spelarna har hittat, skall visas. Programmet skall hålla reda på vilken spelares tur det är och detta skall tydligt markeras i fönstret. Man kan t.ex. använda olika färg i komponenten som visar spelarnas poäng. Ett exempel på hur det kan se ut visas i figuren nedan, i vilken Spelare 1 just har valt två kort som visade sig vara olika. I denna figur används gul bakgrundsfärg för att visa vems tur det är.

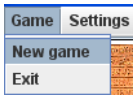


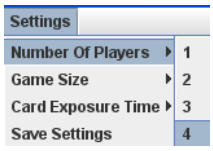
Här ges exempel på några tillägg som gör programmet lite intressantare. **Det är obligatoriskt att göra tilläggen 1 eller 2, samt 3.** (Det rekommenderas förstås att du gör alla tilläggen.)

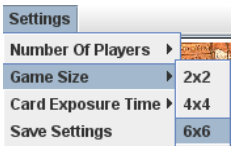
Menyer

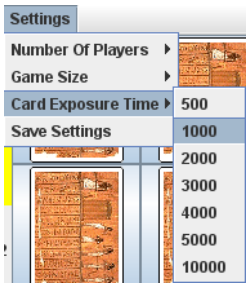
Programmet skall ha två menyer Game och Settings. Menyn Game skall innehålla alternativen New Game och Exit (Obligatoriskt).

För att förenkla för användaren kan det vara lämpligt att begränsa valmöjligheterna något. Exempel på hur det kan se ut visas nedan.









1. Olika antal spelare

Inställningsmenyn Settings skall innehålla alternativet Number Of Players så att man kan ändra antalet spelare. Man skall kunna vara fler än två spelare. Det skall också vara möjligt att köra programmet med endast en spelare. Eftersom det då är meningslöst att räkna hur många par man hittat, skall istället visas hur många gissningar man gjort. Man kan då köra flera gånger i följd och jämföra hur många försök som behövdes för att hitta alla paren.

2. Storlek och tid

Inställningsmenyn Settings skall innehålla alternativen Game Size och Card Exposure Time så att man kan ändra antalet rader och kolumner, samt den tid de båda korten visas när en spelare har klickat på det andra kortet.

3. Kom ihåg inställningar

Lägg till ett menyalternativ med namnet Save Settings i inställningsmenyn. När användaren väljer detta skall de aktuella inställningarna (antal spelare om du gjort tillägg nummer 1 och antal rader och kolumner, samt tidsintervall, om du gjort tillägg nummer 2) sparas i en fil med namnet .memory.config. Varje gång man startar programmet skall det undersöka om filen med inställningar finns. Om så är fallet skall programmet läsa informationen i filen och initiera programmet på det sätt som anges i filen, annars med standardvärdena 2 spelare, 4 x 4 kort, samt 2 sekunders visningstid.

4. Frivilligt eget förslag

Du kanske har några egna idéer om hur programmet kan göras roligare. Diskutera dem med din handledare innan du börjar.

Ett exempel kan vara att visa namnen på spelarna istället för att numrera dem. Man kan också tänka sig ljudeffekter. Om en spelare har hittat två kort som är lika skall det vara ett positivt ljud annars ett negativt ljud. Det går att hitta lämpliga ljudfiler genom att leta efter filer med suffixet .wav.

Exempel

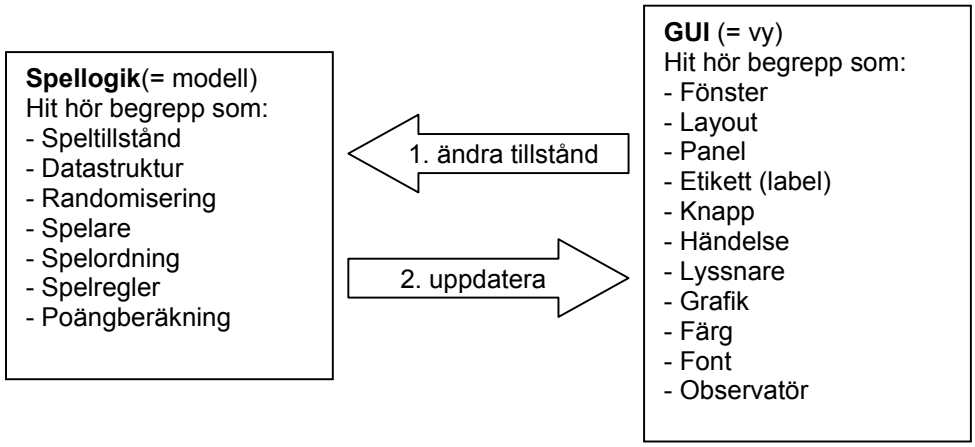
Ett exempel på hur det kan se ut när man kör en utökad version av programmet visas i nedanstående figur.

Designprinciper

Programmet skall delas in i ett antal lämpliga klasser som samarbetar för att lösa uppgiften. Trots att det program ni skall utveckla är relativt litet kan designen av programmet göras på olika sätt, och med varierande grad av objektorienterad ansats. Beroende på hur det struktureras erhålls en mer eller mindre skalbar design (= förmåga att växa utan ”växtvärk”). Vi skall sträva efter att strukturera så att designen får egenskaper som

- Hög *kohesion*: Varje klass och metod skall ha en tydligt definierad och väl avgränsad uppgift och lösa *ett* problem – inte många problem.
- Låg *koppling*: Klasserna skall inbördes ha så lite med varandra att göra som möjligt. Framför allt skall de klasser som handhar själva spellogiken inte direkt känna till de klasser som hanterar det grafiska gränssnittet.

För att uppnå god skalbarhet genom ovanstående egenskaper skall programmets arkitektur utformas enligt designmönstren Model-View-Controller och Observer som gås igenom i samband med momentet om grafiska användargränssnitt. Begreppet *model* (modell) syftar här på de klasser som sköter spellogiken, medan *view* (vy) syftar på klasserna som sköter det grafiska användargränssnittet (GUI). Begreppet *controller* (kontroll) syftar på de lyssnarobjekt som hanterar de händelser som genereras när användaren gör olika saker med gränssnittet, t.ex. trycker på en knapp. När en händelse skett skall oftast tillståndet i ett modellobjekt förändras på något sätt. Genom att låta vyn använda modellen, men isolera modellen från all kunskap om vyn, uppnås låg koppling mellan modellen och vyn. Det gör det enklare att variera det grafiska gränssnittet, t.ex. om man vill göra en PC-version och en mobilapplikation baserade på samma underliggande komponenter för spellogiken. Följande figur tydliggör vilka begrepp som bör hanteras i respektive del i programmet:



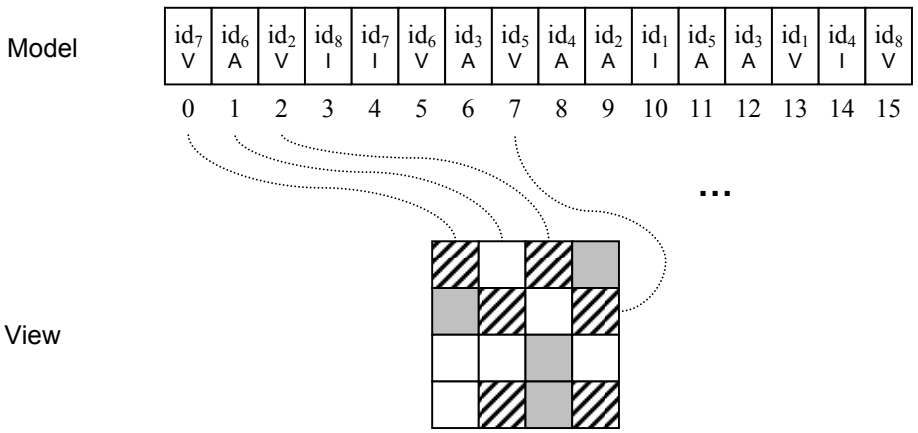
Ex. När användaren trycker på en knapp i fönstret generas en händelse. En lyssnare tar emot händelsen och anropar en metod i ett spellogikobjekt. Metoden förändrar tillståndet i logikobjektet. Logikobjektet uppdaterar en komponent i fönstret (via Observer-mönstret) som ritar om komponenten.

Detaljdesign

En fördel med MVC-mönstret är att man lättare kan frikoppla delarna modell och vy från varandra även rent begreppsmässigt. I vårt exempel skall spelplanen visuellt bestå av ett kvadratiskt rutmönster i ett fönster. Detta innebär inte nödvändigtvis att datarepresentationen i spellogiken bör betraktas som en tvådimensionell struktur. I fönstret implementeras varje kort som en knapp innehållande en bild. Det enda det objektet behöver kunna är att visa fram- resp. baksidan på kortet, alltså två olika bilder, samt en neutral bakgrund som motsvarar att kortet tagits bort från spelplanen.

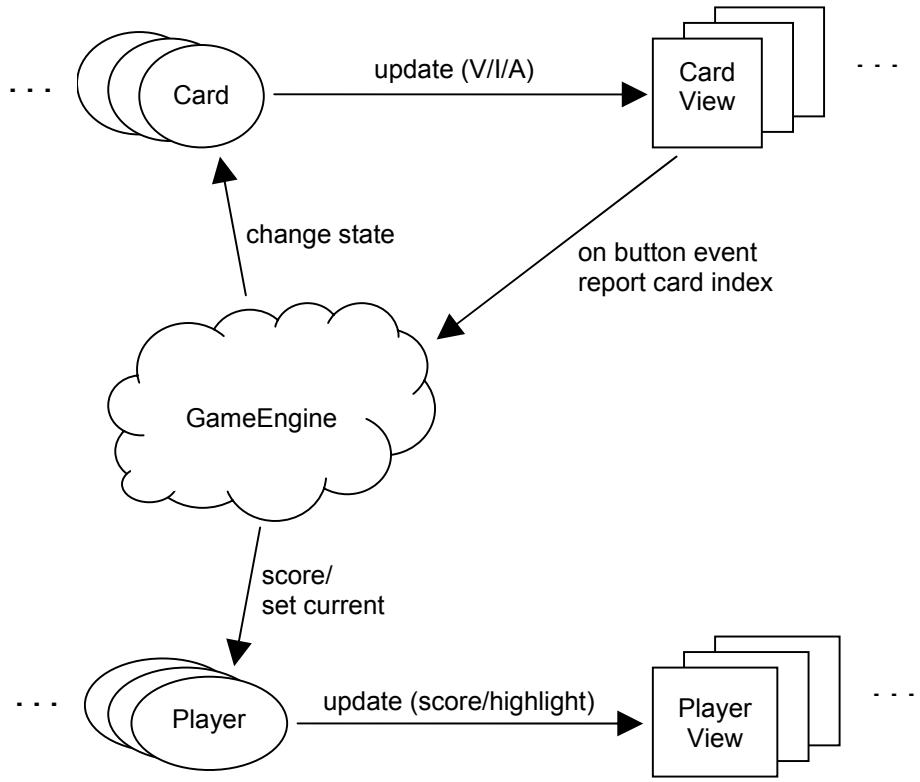
I spelmotorn räcker det om varje korttyp representeras av ett unikt heltal som identifierar vilken bild som skall visas i fönstret när kortet är uppvänt, samt kortets tillstånd (visible (V), invisible (I), absent (A)). Dessa egenskaper representeras av klassen Card. Om korten lagras i ett fält kommer deras position i fältet att logiskt motsvara respektive korts position i fönstret, om vi antar att korten placeras ut radvis från vänster till höger i fönstret.

Ex. Om fönstret skall innehålla 4 x 4 kort med bilder numrerade id_1,…, id_8 så kan modellen representera spelets logiska datainnehåll så här (snedstreckade och gråa rutor motsvaras i verkligheten av bilder):



Spelmotorn kan alltså fokusera på logiken i spelet genom att manipulera kortobjekt och behöver inte bry sig om kortens eventuella grafiska presentation. Spelet handlar ju egentligen i grund och botten om att hitta par i en heltalsvektor! Kopplingen mellan kort och kortens visuella presentation sker med Observer-mönstret. Varje kort observeras av en kortvy. Så snart kortets tillstånd förändras så skall kortvyn avspegla förändringen genom att ändra sitt utseende.

Samma typ av relation bör gälla mellan modellklassen Player som håller reda på information om en spelare, och komponenten PlayerView i vyn som visar information om spelarens identitet och poäng, se nästa figur.



Utplacering av kort

Korten skall väljas slumpmässigt ur en given mängd och dessutom placeras slumpmässigt i fönstret. Det skall finnas två av varje kort. Antalet kortpositioner i fönstret (N) måste alltså vara jämnt. Om antalet tillgängliga kortbilder betecknas med K måste alltså följande villkor vara uppfyllda

$$N \bmod 2 = 0$$
$$N \leq 2K$$

Villkoren skall kontrolleras av programmet och lämpligt undantag kastas om något av dem ej är uppfyllt. Ex. för en spelplan med 4 x 4 kort behövs minst 8 olika bilder. Spelet blir förstås lite svårare om man har många bilder eftersom det då kommer att dyka upp bilder i varje ny spelomgång som inte varit med på ett tag.

Att skapa ett slumpmässigt urval av bildnummer kan göras på följande sätt:

1. Skapa en lista med talen 0,…,K-1 som element.
2. Randomisera ordningen mellan listelementen.
3. Placera två av varje av de N/2 första elementen i en ny lista.
4. Randomisera ordningen mellan listelementen i den nya listan.

Det nu lätt att skapa fältet av kort enligt det tidigare exemplet. Det blir ett fält med N element. I varje fältelement lagras ett Card-objekt innehållande ett bild-ID hämtat från motsvarande position i listan från steg 4 ovan. Varje kort ges grundtillståndet INVISIBLE. Använd lämpligen listor av typen ArrayList och utför randomiseringen med metoden Collections.shuffle.

Ex. Antag att vi har 20 olika bilder och ett spel med 4 x 4 kort.

1. Skapa den första listan av alla möjliga bild-ID:n:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

2. Randomisera listan:

14 10 16 2 18 4 12 6 19 8 1 11 5 13 0 15 3 17 9 7

3. Placera två av varje av de åtta första elementen i en ny lista:

14 14 10 10 16 16 2 2 18 18 4 4 12 12 6 6

4. Randomisera den nya listan:

2 12 4 14 10 18 6 16 2 18 4 14 12 6 16 10

Utveckling av programkoden

Eftersom programmet struktureras enligt MVC-arkitekturen är de två delarna modell och vy relativt oberoende av varandra. De kan utvecklas i valfri ordning, eller båda samtidigt. Består labbgruppen av två personer kan det senare vara värt att prova. Metodanropen i modellobjekten som har med Observer-mönstret att göra kan läggas in tidigt. Ett anrop av notifyObservers kan ju göras oberoende av om några observatörer finns eller ej. Modellklasserna testas enklast genom att låta metoderna skriva ut lämplig information på skärmen, t.ex. för att kontrollera att rätt kort konstrueras. GUT:t kan göras nästan klart.

Utnyttja gärna *stubbar* vid utvecklingen. Behövs en klass som inte finns än, för att testa en annan klass, kan man göra en provisorisk klass med mer eller mindre tomma metoder (=stubbar). För att testa GUI:t kan man t.ex. göra en modellklass som simulerar spellogiken på ett enkelt sätt.

Modellklasser

Klassen Card

Lagrar ett bild-ID samt ett tillstånd. Tillståndet skall vara av typen State som är en s.k. uppräkningstyp. Den deklareras i klassen med

```
public static enum State { VISIBLE, INVISIBLE, ABSENT }
```

När man vill använda ett värde i typen skriver man t.ex. State.VISIBLE.

- Klassen skall ärva från Observable.
- Det skall finnas metoder för att läsa av och ändra tillståndet.
- Det skall finnas en metod för att läsa av bild-ID.

Klassen Player

Lagrar en spelares identitet och poäng. Identiteten kan vara ett nummer eller spelarens namn i form av en sträng.

- Klassen skall ärva från Observable.
- Det skall finnas metoder för att läsa av och öka spelarens poäng.
- Det skall finnas en metod för att läsa av spelarens identitet.
- Det skall finns en metod för att ange om det är spelarens tur eller ej.

Klassen GameEngine

Detta är huvudklassen i modellen.

- Klassen skall skapa ett fält av kortobjekt enligt ovan beskriven procedur, samt ett fält av spelarobjekt.
- Klassen tar emot information om vilket kort användaren valt och uppdaterar relevant information.
- Klassen ansvarar för att vyn uppdateras när något behöver ritas om.
- Konstruktorn bör ha inparametrar för antalet spelare, antalet olika bilder (K), antalet kort i den aktuella spelomgången (N), samt den önskade exponeringstiden för korten. Om K och N inte uppfyller tidigare diskuterade villkor skall undantaget IllegalArgumentException kastas.
- Eftersom man skall kunna välja att starta ett nytt spel utan att starta om programmet är det lämpligt att införa en metod som skapar ett ny spelomgång. Denna kan anropas från konstruktorn.

  public void newGame(*parametrar*)

- Klassen bör också ha en metod som vyn kan anropa för att meddela vilket kort användaren valt:

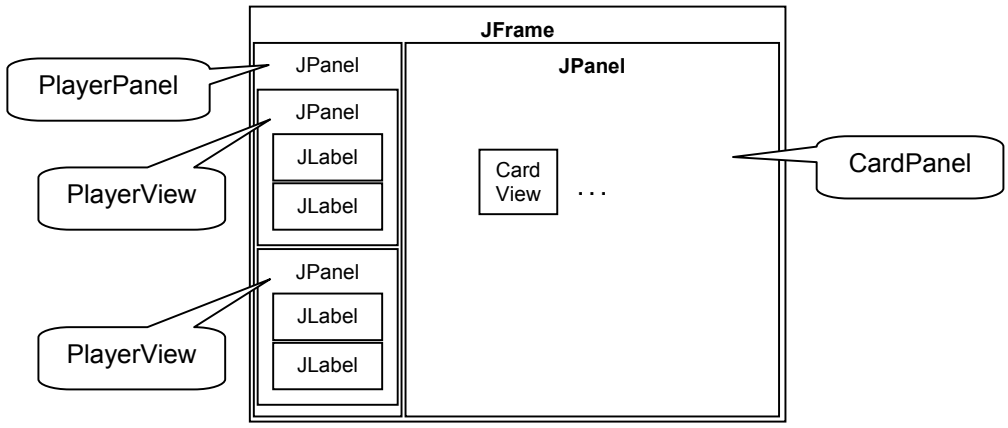
  public void selectCard(int index)

  Metoden skall registrera relevant information som påverkas av valet: Var det första eller andra kortet? Skall poäng utdelas till spelaren? Behöver något uppdateras i vyn? Skall spelaren få fortsätta, eller är det nästa spelares tur? Etc.
- Det skall finnas en metod för att registrera CardView som observatör på Card.
- Det skall finnas en metod för att registrera PlayerView som observatör på Player.

Vyklasser

Klassen Gui

Klassen är programmets huvudklass. Den skapar fönstret och ser till att alla nödvändiga objekt skapas. Fönstret skall byggas upp hierarkiskt med hjälp av paneler. Vid en ny spelomgång kan dessa enkelt tas bort och bytas mot nya. Klasserna CardView , CardPanel, PlayerView, samt PlayerPanel beskrivs längre ner.



- Konstruktorn skall ladda inställningarna, läsa in bildfilerna, skapa fönstret samt en menyrad i fönstret.
- Det skall finnas en metod loadSettings som läser in inställningarna från en fil. Om inga sparade inställningar hittas sätts standardvärden enligt följande: antal spelare 2, spelstorlek: 4 x 4, visningstid: 2 sek. I en första version kan du hoppa över loadSettings och sätta standardvärdena direkt.
- Det skall finnas en metod newGame som skapar ett objekt av klassen GameEngine, samt panelerna enligt ovan.

Gör först en preliminär version utan inläsning av inställningar och utan GameEngine. Testa och bygg ut efter hand.

Klassen CardView

Klassen ansvarar för den visuella presentationen av ett kort i fönstret.

- Klassen skall ärva från JButton och implementera gränssnittet Observer. Eftersom klassen ärver från JButton blir den en komponent som kan placeras i en panel.
- CardView skall lagra en bild i form av ett objekt av klassen ImageIcon.
- Varje CardView-objekt skall observera ett Card-objekt och förändra sitt utseende beroende på det objektets tillstånd.

Klassen CardPanel

Klassen lagrar ett antal CardView-objekt.

- Klassen skall ärva från JPanel.
- Det skall finnas en konstruktor med åtminstone följande parametrar:
    - en referens till GameEngine.
    - ett fält av typen ImageIcon[] med bilder på kortens framsidor.
    - antalet rader.
    - antalet kolumner.
    - Om ni väljer att låta korten ha en baksidesbild (som i exemplen ovan) skall det finnas en motsvarande parameter för denna.
- Angivet antal kortvyobjekt av typen CardView skall skapas. Varje kortvyobjekts bild-ID hämtas från GameEngine-objektet. För att kunna göra det måste aktuell rad och kolumn räknas om till ett index.
- Varje kortvyobjekt skall ha en lyssnare som vid knapptryck rapporterar objektets index till GameEngine med anrop av metoden selectCard.
- Varje CardView-komponent skall registreras som observatör på motsvarande Card-objekt.

Klassen PlayerView

Klassen skall sköta visningen av information om en spelare.

- Klassen skall ärva från JPanel och implementera gränssnittet Observer.
- Spelarens identitet skall visas ovanför spelarens poäng, som framgår av figurerna på sid. 2-3.
- Spelarens identitet kan representeras av ett nummer eller ett namn, välj själv.
- Visa informationen i två Swing-komponenter av typen JLabel.
- Det skall finnas en metod

  ```
  private void highLight(boolean on)
  ```

  som sätter olika bakgrundsfärg beroende på parameterns värde.
- När klassens update-metod får ett objekt av klassen Player som argument skall den andra parametern innehålla antingen ett booleskt värde som anger hur bakgrundsfärgen skall ändras, eller ett heltal som anger spelarens nya poäng. (Detta måste förstås lösas på motsvarande sätt i GameEngine.)

Klassen PlayerPanel

Klassen lagrar ett antal PlayerView-objekt.

- Klassen skall ärva från JPanel.
- Det skall finnas en konstruktor med två parametrar.
    - en referens till GameEngine.
    - ett heltal som anger antalet spelare.
- Det skall skapas och adderas ett antal PlayerView-komponenter enligt angivet antal.
- Varje PlayerView-komponent skall registreras som observatör på motsvarande Player-objekt.

Hjälpklasser

Här beskrivs ett par klasser som sköter programmets kommunikation med filsystemet. De beskrivs separat eftersom de inte faller inom MVC-konceptet.

Klassen Config

Lagrar programmets inställningar för t.ex. antal spelare, spelstorlek, och tidsfördröjning vid kortvisning. Inför två konstruktorer, en med parametrar, och en parameterlös som sätter standardvärden på egenskaperna.

Klassen FileIO

Klassen sköter inläsning av bildfilerna samt hämtar och sparar programmets inställningar. Följande metoder skall finnas i klassen:

- En metod för att läsa in bilder för kortens framsidor.

  ```
  public static ImageIcon[] loadUpsideImages()
  ```

  se vidare nedan hur inläsningen kan göras.
- En metod för att spara programmets inställningar i en fil. Välj själv lämplig metod: textfil eller objektfil. Fördelen med det senare är att den *inte* går att redigera. Fördelen med den förra är att den *går* att redigera. ☺

  ```
  public static void saveSettings(Config settings)
  ```

- En metod för att hämta sparade inställningar till programmet.

  ```
  public static Config loadSettings()
  ```

Det finns gott om lämpliga bilder att leta upp på Internet, eller använd de som finns på kurshemsidan. Där ligger en mapp med bilder på vanliga spelkort. Mappen carddeck/cards-1-52 innehåller 52 GIF-bildfiler med kortens framsidor (*lägg inga andra filer i denna katalog!*). Filen carddeck/backside.gif innehåller en bild på kortens baksida. När man läser en bildfil är det praktiskt att använda klassen ImageIcon. Sådana objekt kan ju dessutom användas i olika GUI-komponenter. Mediafiler läses i en speciell tråd. ImageIcon använder en s.k. MediaTracker för att invänta att läsningen avslutas på rätt sätt.

Ex. Läs in bildfilen filur.gif till ett ImageIkon-objekt:

```
ImageIcon pic = new ImageIcon(”filur.gif”);
```

För att läsa in alla kortbilderna till programmet kan man använda standardklassen File. Man börjar med att skapa ett File-objekt som hanterar bildmappen i filsystemet:

```
File picdir = new File(bildmapp)          // namnge bildmappen enligt ovan!
```

Därefter kan man skapa ett fält av File-objekt där varje objekt hanterar en bildfil:

```
File[] picfiles = picdir.listFiles();
```

För varje element i fältet picfiles kan man sedan anropa metoden getPath för att få reda på namnet på motsvarande bildfil. Filnamnen används för att skapa fältet med ImageIcon-objekt som skall returneras av metoden loadUpsideImages ovan.

Allmänna tips

När man konstruerar program med grafiska gränssnitt kan man ibland få problem med att gränssnittet inte hinner med att rita upp allting när programmet startas. Fönstret kanske ritas upp rätt, men det grafiska innehåll som beror av tillståndet i modellobjekten kommer inte med. Om vi programmerar enligt MVC-modellen så måste av naturliga skäl vissa modellobjekt skapas innan vyobjekten eftersom de senare använder modellobjekten. Man kan då inte placera uppdateringsanrop för vyn i modellobjektens

konstruktorer eftersom sådana anrop då sker innan vyn är beredd att behandla dem. En lösning är att införa en speciell initieringsmetod i modellen som vyn kan anropa när fönstret är uppritat. I princip kan det se ut så här:

```
Model m = new Model();
View v = new View(m);
m.addObserver(v);
m.init();                    // init uppdaterar v
```

Samtliga steg kan rent tekniskt även placeras i View, men principen framgår förhoppningsvis. I vårt spel kan t.ex. GUI när fönstret är skapat anropa init i spelmotorn som då uppdaterar alla kortobjekten så att de visar rätt bild. Om detta inte sker kommer GUI:t inte att visa några kortbilder förrän respektive kort väljs av en spelare.

Standardklasser och metoder

Här följer ett urval av metoder och klasser som är användbara i lösningen. Läs dokumentationen i Java API, även om ärvda metoder. De flesta GUI-klasserna ärver hierarkiskt i flera led från ett antal basklasser. Att bara titta i den aktuella klassen räcker sällan.

java.util.
ArrayList
Iterator
Collections
  shuffle()
Observer
  update()
Observable
  addObserver()
  setChanged()
  notifyObservers()

javax.swing.
Timer
  start()
  setRepeats/()
JFrame
  setTitle()
  setLayout()
  setJMenuBar()
  add()
  setLocationRelativeTo()
  setDefaultCloseOperation()
  setVisible()
  pack()
JPanel
  setLayout()
  add()
JLabel
  setText()
  setOpaque()
JButton
  setIcon()
  addActionListener()
  setBackground()
  setEnabled()
JMenuBar
  add()
JMenu
  add()
JMenuItem
  addActionListener()
ImageIcon

java.awt.
GridLayout

java.awt.event.
ActionListener
  actionPerformed()
ActionEvent

java.io.
File
  listFiles()
  getPath()
FileInputStream
DataInputStream
ObjectInputStream
FileOutputStream
DataOutputStream
ObjectOutputStream