Slutrapport för Den försvunna kossan

Ingrid Stake, Olivia Månström, Tove Nilsson, Elvina Fahlgren

24 oktober 2021
Version 1

1 Introduktion

”Den försvunna kossan” är ett sällskapsspel i form av en desktopapplikation. Spelet går ut på att hitta en ko, som gömmer sig under någon av spelets alla brickor som är placerade runt om på spelplanen, och ta sig i mål. Varje spelare turas om att slå en tärning och förflyttar sin spelpjäs så många steg som tärningen visar. När spelaren hamnar på en bricka kan spelaren välja att betala eller slå tärningen för att öppna denna. Under en bricka gömmer sig en kobjällra, komocka, ko, häst, gris eller ingenting. Varje bricka en specifik egenskap och påverkan på spelaren.

1.1 Definitioner, akronymer, och förkortningar

- JSON - är textbaserat format som beskriver ett objekt och används för att utbyta data. Det är enkelt för människor att skriva och läsa, samt enkelt för en dator att analysera.
- GSON - ett Java bibliotek som bland annat kan användas till att konvertera en JSON sträng till motsvarande Javaobjekt.
- JUnit - För enhetstestning av Java kan ramverket JUnit användas.
- Modell - utgör logiken av den information som spelet bearbetar.
- Vyn - Exponerar delar av modellen för användaren.
- Kontroller - Tar emot input från användaren för att trigga förändring i vyn eller modellen.

2 Krav

2.1 Användarberättelser

1. Som användare vill jag att det finns spelare för att kunna spela spelet.

   #### Acceptanskriterier

   - Spelaren ska ha en egen färg.
   - Det ska vara 1-4 spelare.

   #### Uppgifter

   - Skapa en spelare-klass.
   - Skapa en boolean som avgör om spelaren har hittat kossan.
   - Skapa en boolean som avgör om spelaren har hittat ett visum.
   - Skapa en metod som flyttar spelare till vald position.
   - Skapa en int för att hålla koll på spelarens saldo.

2. Som användare vill jag att det finns en spelplan för att få en överblick över spelet.

   #### Acceptanskriterier

   - Spelplanen består av positioner av olika typer.
   - Positionerna ska uppbyggd som en graf.

   #### Uppgifter

   - Skapa en JSON fil att representera spelplanen och alla positioner.
   - Kunna läsa in en JSON fil.
   - Hantering för omvandling av JSON fil till Java objekt.
   - Felhantering för syntaxfel i JSON fil.
   - Lägg till noder och dess grannar i JSON filen.

3. Som användare vill jag att spelarna visuellt skiljer sig åt för att veta vem som är vem.

Acceptanskriterier

- Det ska finnas en spelfigur för vardera spelare.

Uppgifter

- Skapa olika FXML-filer med olika färger för de olika spelarna.
- Skapa en Enum att representera olika färger på spelare.
- Skapa en metod i klassen PlayerColor som beräknar vilken FXML-fil som hör till vilken spelarfärg.

4. Som användare vill jag ha olika positioner på spelplanen utan/med markörer för att kunna ta mig fram i spelet

Acceptanskriterier

- Det ska vara rätt fördelat med markörer på spelplanen.
- Markörerna går att vända upp.
- Samtliga markörer ska ha en egen FXML-fil.

Uppgifter

- Skapa Enum för olika typer av spellogik för positioner.
- Skapa FXML-filer för framsidan av varje markör.
- Skapa en klass TilePosition för positionerna av markörer.
- Skapa en metod som uppdaterar markörens läge i TilePosition-klassen.

5. Som användare vill jag att spelpjäsen förflyttar sig till den position jag valt för att spelet ska kunna fortgå.

Acceptanskriterier

- Spelpjäsen ska förflyttas till den valda positionen.
- Endast den aktiva spelaren ska kunna förflytta sig.

Uppgifter

- Ett tryck på en nod ska starta ett event.
- Kontrollen ska bli medveten om att en användare har tryck på en nod.

- Kontrollen ska anropa rätt metod från modellen.

6. Som användare vill jag kunna slå en tärning för att ta mig fram i spelet.

Acceptanskriterier

- Spelaren ska kunna rulla tärningen.
- Efter tärningen är rullad ska spelaren kunna förflytta sig.
- Spelaren ska kunna rulla tärningen igen om hen har förflyttat sig till en markör, om inte ska spelaren bara kunna rulla tärningen en gång.

Uppgifter

- Skapa en tärningsklass.
- Skapa en metod som slumpar ett tal mellan 1-6 och returnera talet.
- Gör tärningskastet aktivt för att förflytta sig på spelplanen eller för att öppna en markör.
- Tillåt den aktiva spelaren att rulla tärningen.
- Visa resultatet av tärningskastet.

7. Som användare vill jag att pjäsen förflyttas till den position jag väljer så länge det är inom det upplysta området för att spelet ska följa reglerna.

Acceptanskriterier

- Spelpjäsen försvinner från den ursprungliga positionen och dyker istället upp på den valda positionen på spelplanen.
- Spelpjäsens point-property uppdateras.
- Upplysningen försvinner från valbara positioner.

Uppgifter

- Skapa en FXML-fil för att visa att noden är upplyst.
- Skapa en knapp för simulering av tärningskast.
- Byt till den upplysta FXML-filen för de positioner som är giltiga efter ett tärningskast.
- Byt tillbaka till den vanliga FXML-filen när förflyttningen har skett.

8. Som användare vill jag ha en startposition för att inleda och avsluta spelet.

   #### Acceptanskriterier

   - Det ska finnas en startposition på spelplanen.

   #### Uppgifter

   - Skapa logik för för att spelet ska avslutas när en spelare som har antingen kossan eller ett visum tar sig till startpositionen.

9. Som användare vill jag välja antalet spelare innan påbörjat spel för att kunna anpassa spelet.

   #### Acceptanskriterier

   - Applikationen ska kunna läsa input från spelarna.
   - Användarna kan välja bland antalet spelare.
   - Spelet kan inte starta förens antalet spelare är bestämt.
   - Antalet spelare kan inte ändras när spelet har börjat.
   - Spelet anpassas efter antalet valda spelare.

   #### Uppgifter

   - Begränsa antalet spelare till 4.
   - Skapa en *choise box* där användaren kan välja antalet spelare.
   - Koppla antalet spelare som ska spela till den valda siffran.

10. Som användare vill jag kunna starta spelet för att spela spelet

    #### Acceptanskriterier

    - Spelet startar när spelaren har tryckt på startknappen.
    - Spelarna tilldelas rätt mängd pengar.

Uppgifter

- Dela ut startbelopp till varje spelare.
- Placera spelarna på startnoden.
- Skapa en klass som hanterar vybyte.
- Skapa en klass som hanterar vyn för startsidan.
- Skapa en FXML-fil för startvyn.
- Koppla kontroller till knapp på startvyn.

11. Som användare vill jag att markörerna får en slumpad position på spelplanen för att det ska vara spännande att spela.

Acceptanskriterier

- Markörerna ska få en slumpad position på spelplanen.

Uppgifter

- Skapa en metod i spellogiksenumen som returnerar en slumpad lista med rätt fördelning av spellogikselement på spelplanen.

12. Som användare vill jag kunna vända upp en markör genom att slå tärningen för att spelet ska fortgå.

Acceptanskriterier

- En markör kan endast vändas upp vid tärningsslagen 4, 5 och 6.
- Markören ska vändas och det ska vara synligt för spelaren vad som låg under.

Uppgifter

- Skapa en gamelogic klass.
- Om spelar får ett giltigt tärningsslag (4,5,6) vänds markören upp.
- Om ja, kör metoden *executeTileTurn*, om nej händer inget.

13. Som användare vill jag kunna vända upp en markör genom att betala pengar för att spelet ska fortgå.

Acceptanskriterier

- Ska endast vara ett alternativ om spelaren har tillräckligt mycket pengar.
- Spelaren ska kunna välja att betala pengar för att öppna en markör.
- Markören ska visas för spelaren.

Uppgifter

- Kolla om spelaren har tillräckligt med pengar.
- Debitera spelaren.
- Ändra status på *tilePositionen* så att den är vänd.

14. Som användare vill jag kunna välja att betala 3000kr om jag står på en flygplats för att flyga. (Inte implementerad - brist på tid)
15. Som användare vill jag kunna välja att betala 1000kr om jag står på en båtplats för att åka båt. (Inte implementerad - brist på tid)
16. Som användare vill jag ha spelarkort bredvid spelplanen som visar den aktuella spelaren tillsammans med dennes egendomar för att kunna lägga upp en spelstrategi.

Acceptanskriterier

- Rutorna ska synas under hela spelets gång.
- Rutorna ska visa vems tur det är i spelet.

Uppgifter

- Skapa FXML-filer för respektive spelarkort.
- Skapa en motsvarighet till *ViewResource* för en *playerCardView* som innehåller den aktuella spelaren.
- Spara spelkorten i gameboardview.
- Visa kossa/kobjällra om spelaren har hittat dessa.
- Ladda in playerCardViews i gameBoardView.
- Gör så att currentPlayer-kortet skjuter upp och sedan återgår till sin originalposition när turen går vidare.
- Uppdatera fälten i varje kort när något har skett.

17. Som användare vill jag att spelet avslutas då den första spelaren med kossan/visum tar sig in i mål.

    #### Acceptanskriterier

    - Spelet verifierar att den som gått i mål har ett visum/kossa.
    - Spelomgången avslutas så att inga fler drag kan göras.

    #### Uppgifter

    - Undersök om det är en giltig vinst. Alltså att användare är på startpositionen och har antingen visum eller kossa.
    - Avsluta spelomgången när någon vunnit.

18. Som användare vill jag få en bekräftelse i form av en ny vy om någon vinner för att veta om spelet är avslutat

    #### Acceptanskriterier

    - Utlöses när första användaren kommer i mål med ett visum/kossa.
    - Deklarerar för alla spelare vem som vunnit spelomgången.

    #### Uppgifter

    - Visa vem som vunnit, dess spelkort hamnar i mitten av spelplanen.

2.2 Definition av slutfört

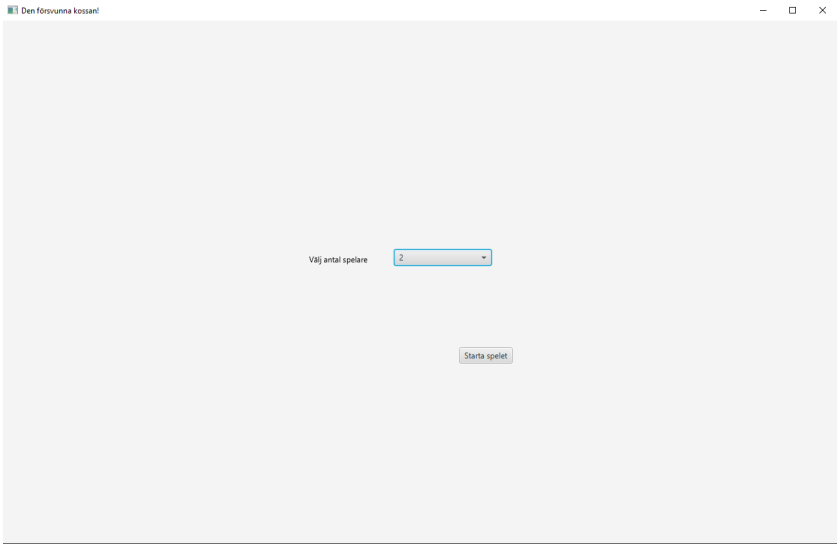
För att en användarberättelse ska anses vara slutförd krävs det att nedan listade acceptanskriterier är uppfyllda.

- Minst två medlemmar i gruppen ska gå igenom koden för att se till att den följer den kodstandard som bestämts.
- Javadoc ska skrivas på kod som ingår publika metoder, samt den kod som kan vecka frågetecken.
- Det ska vara minst 90 % testrapportering av raderna.
- Alla uppgifter för användarberättelsen ska vara gjorda.
- Travis ska godkänna koden.

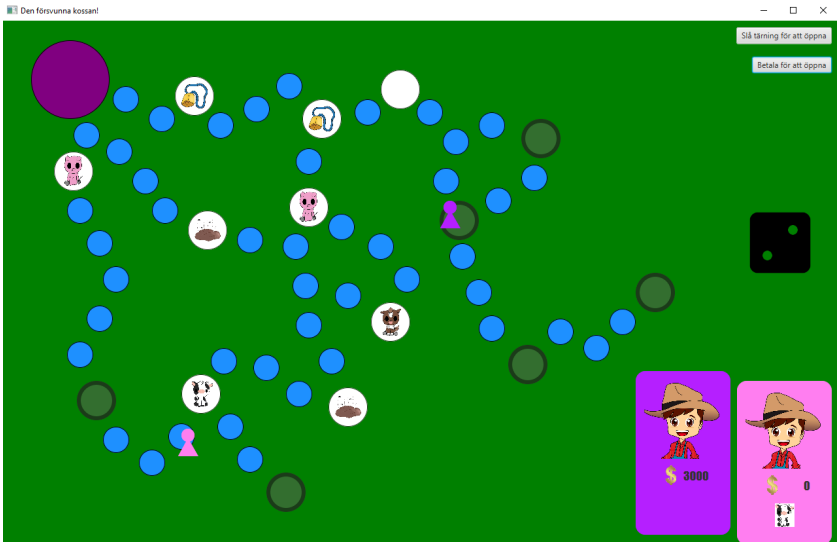
- Användarberättelsen ska finnas representerad i design-modellen, samt beskrivas i SDD och RAD.

2.3 Användargränssnitt

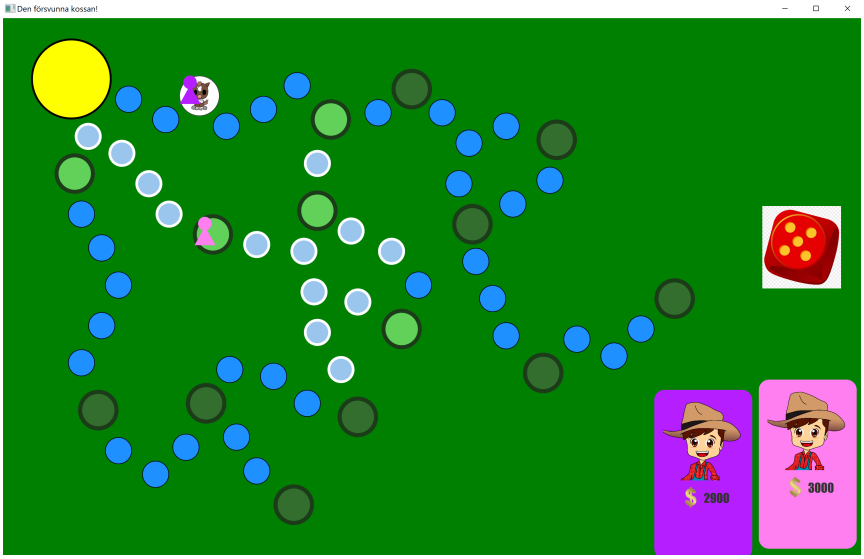
- Startsidan - När användaren startar spelet presenteras en startsida där de har möjlighet att välja antalet spelare.



- Spelplanen - När användaren har valt antalet spelare visas själva spelplanen, i det här fallet är två spelare valda. Här har användaren möjlighet att slå en tärning och ta sig fram på spelplanen. När en spelare står på en markör har den valet att öppna markören genom att betala eller genom att slå tärningen.

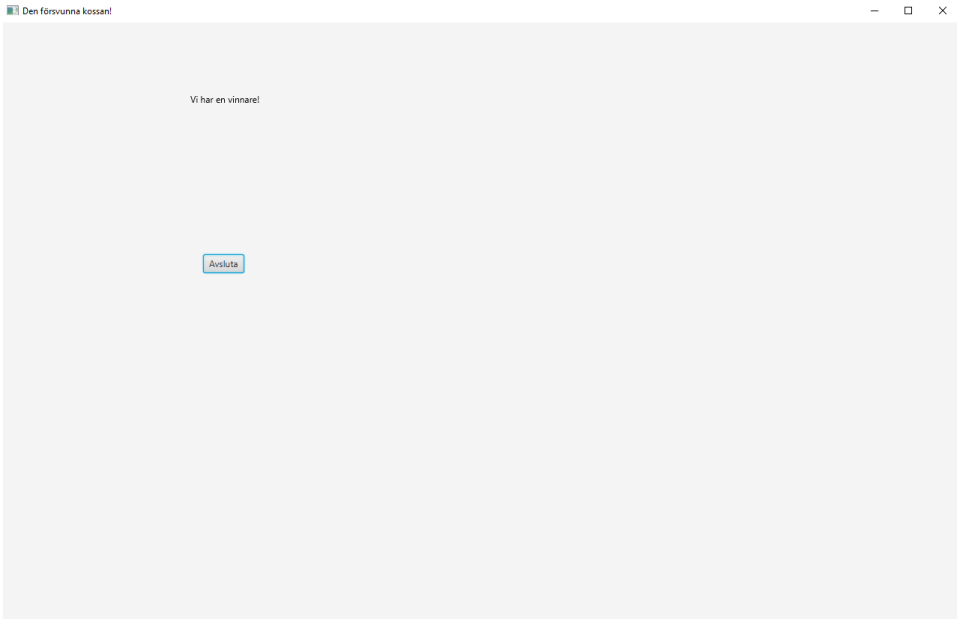


- Upplysta noder - När användaren slår tärningen visas vilka positioner man kan förflytta sig till genom att de lyses upp.

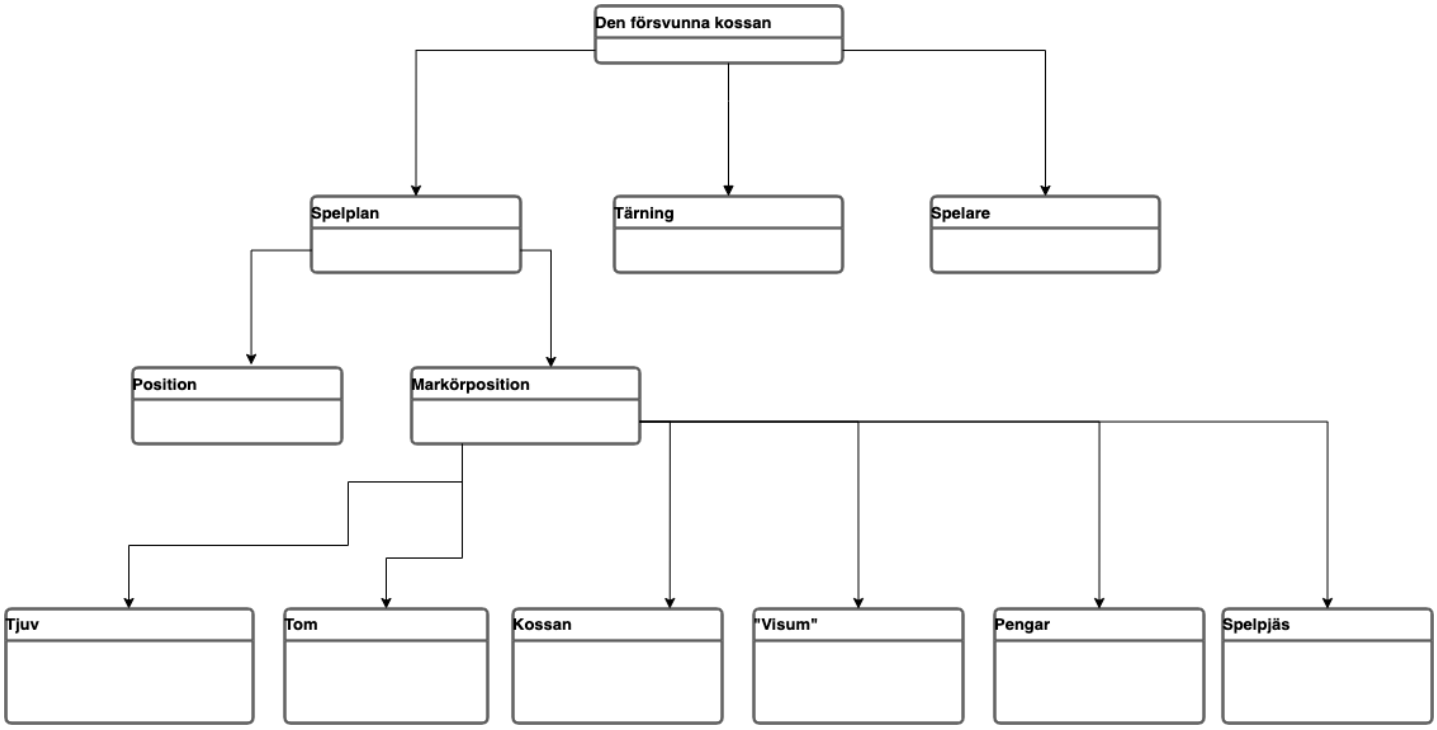


- Spelarkort - På spelplanen visas spelarnas kort där saldo och eventuella egendomar finn synliga för spelarna. Spelkortet för den aktuella spelaren höjs upp för att det enkelt ska synas vems tur det är.

- Slutsidan - När en spelare har tagit sig in i mål visas en sida som talar om att en spelare har vunnit och att spelet har avslutats.



3 Domänmodell



Figur 1: Domänmodell

3.1 Klassansvar

- GameLogic - Ansvarig för logiken under spelets gång.
- Spelplan - Ansvarig för att samordna spelare och positioner.
- Tärning - Ansvarig för att simulera en tärning, det vill säga att slumpa fram värden till resren av programmet.
- Spelare - Den modul med ansvaret att representera en spelare och dess tillstånd.
- Position - Dn modul med ansvaret att representera positioner och dess data och logiktyp.
- MarkörPosition - En modul som representerar markörers positioner och dess data och logiktyp.
- Markörer (Tjuv, Blank, Visum, Pengar, Spelpjäs) - Tillåter spellogiken att bestämma vad som ska ske.

4 System arkitektur

När applikationen startas inleds flödet av startmetoden i *HelloApplication*. Först skapas en instans av *GameModel*, därefter skapas en *GameController* med modellinstansen som argument. *GameController* startar en vy som tillåter användaren att ange antalet spelare och därmed anpassa modellen. I och med att *GameModel* är inparameter kan både kontrollen och vyn lägga till sig själva och sina komponenter som *listeners/observers*. Det innebär att komponenterna vyn och kontroller kan göras medvetna om förändringar i de delar av modellen de prenumererar på utan att modellen behöver vara medveten om de specifika komponenterna. Alternativet till detta hade varit att ha en tidsberoende uppdateringsmetod hos kontrollen respektive vyn.

Efter att spelet har startats sker inget förrän användaren slår tärningen för första gången. Därefter utlöses olika typer av event genom knapptryck som behandlas av modellen och låter spelet fortgå.

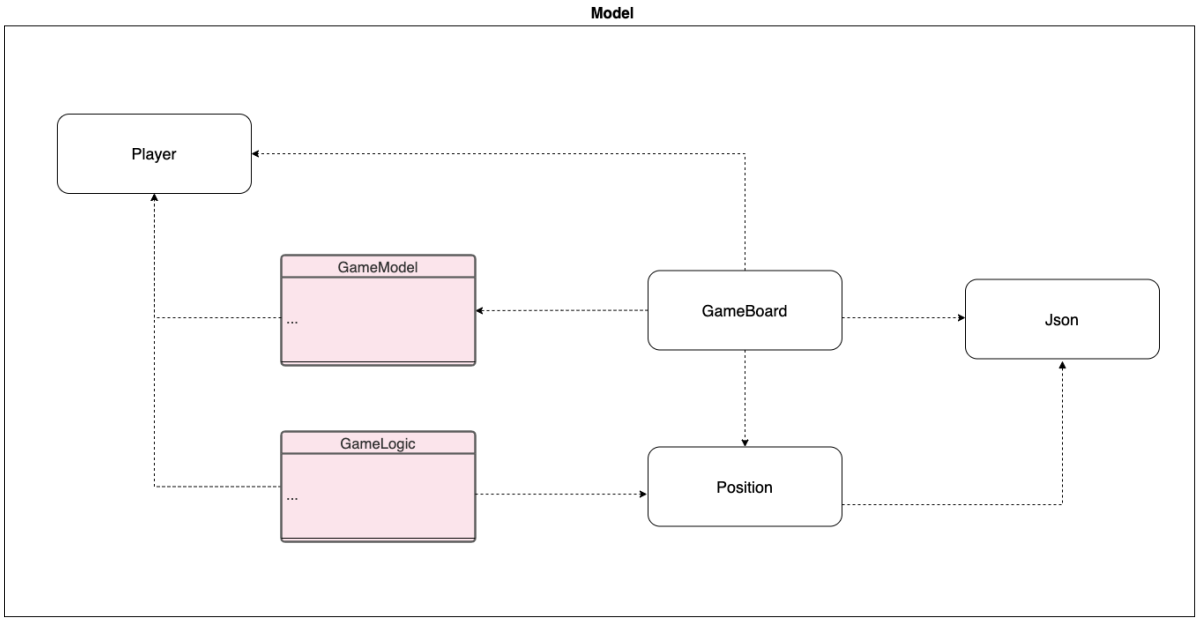
När modellen initierats från kontrollen skapas en spelplan utifrån en JSON-fil, samt rätt antal spelare. Detta sker genom klassen *JSONHandler* som använder sig av GSON för att konvertera JSON-filen till ett javaobjekt, i detta fall *JSONBoardReader*.

5 Systemdesign

5.1 Allmänt

Det kan poängteras att JSON-filen som bygger upp spelplanen är lätt att byta ut, med andra ord kan spelet utökas med flera olika spelplaner att välja mellan. Detta skulle inte påverka spelogik eller övrig mjukvara.

5.2 Relation mellan paket



Figur 2: Designdiagram av kopplingen mellan paketen i modellen.

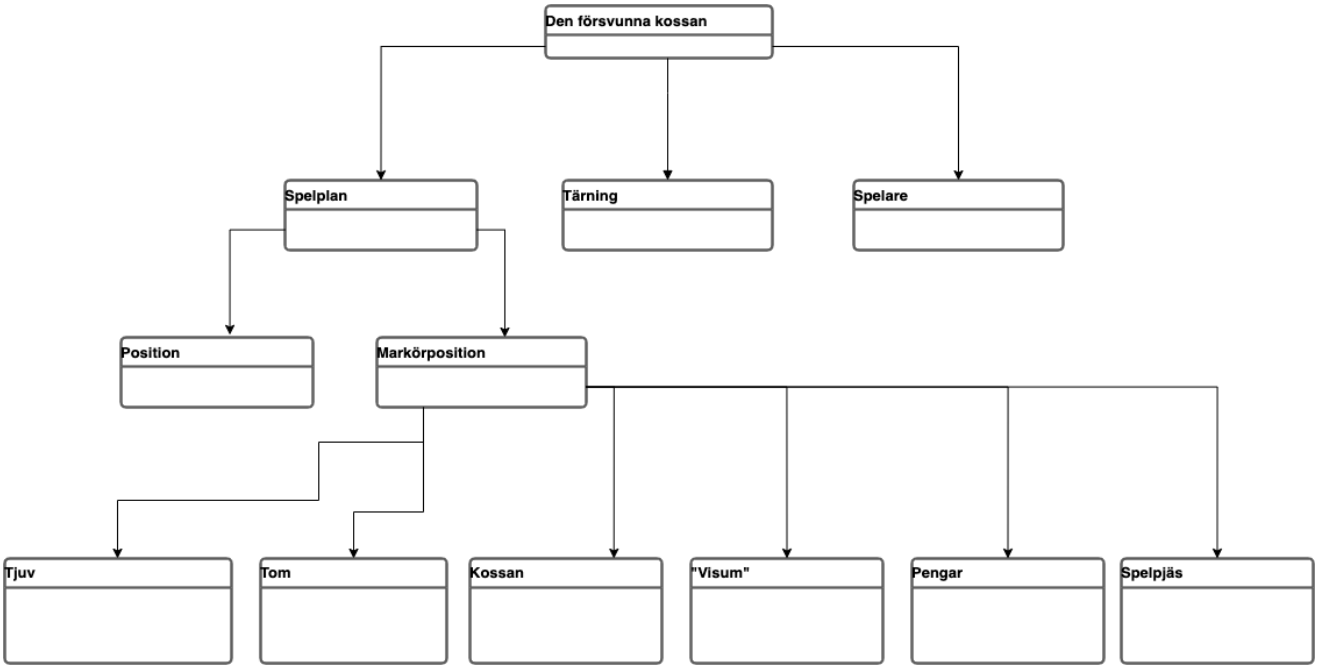
Paketet *Model* består av fyra paket som har ansvar för all logik för spelet, där vardera paket representerar en komponent i applikationen.

Paketet *Player* representerar en spelare och håller information om varje spelare i spelet. Exempelvis vilken färg, hur mycket pengar spelaren har samt huruvida denne hittat kossan eller ett visum eller ej.

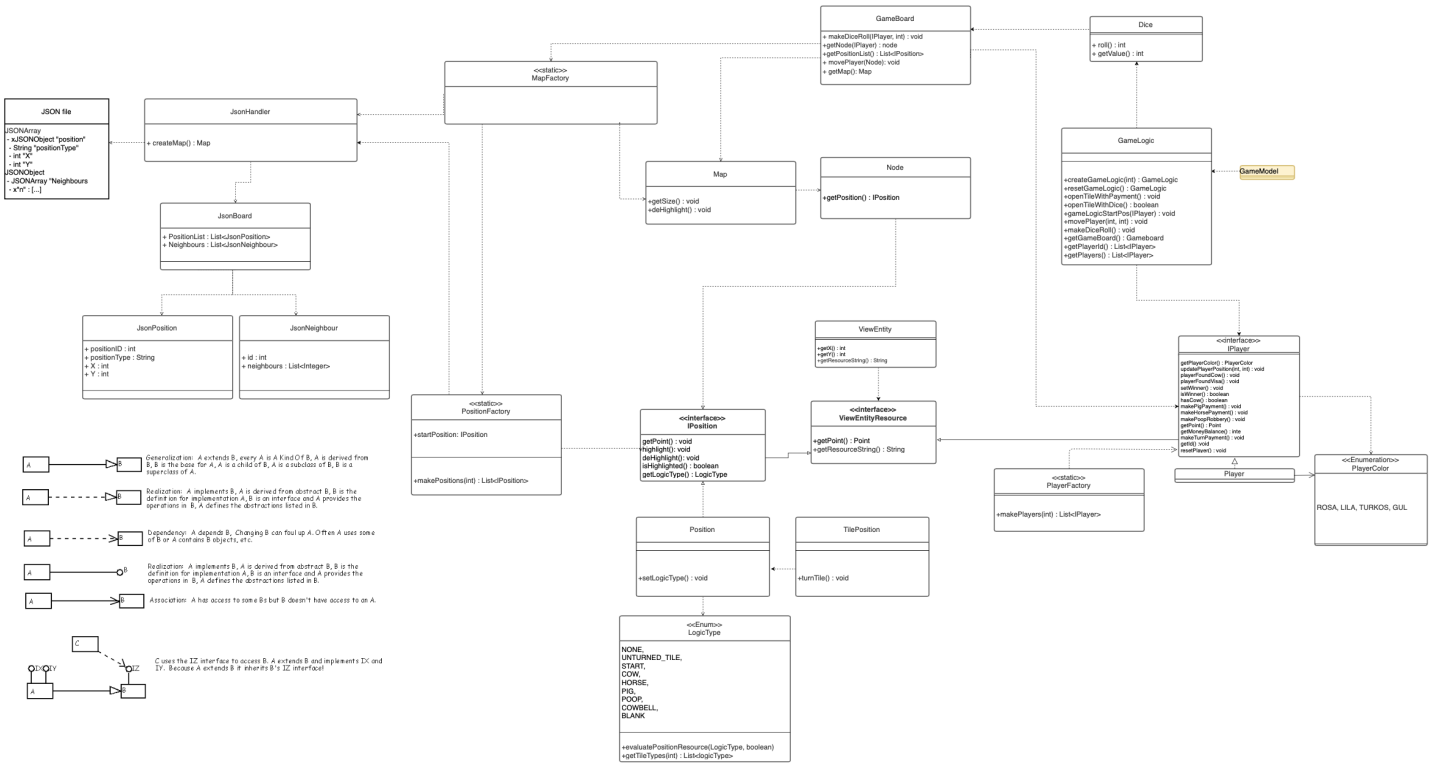
Paketet *GameBoard* representerar en spelplan i spelet. *GameBoard* använder paketet *JSON* för att skapa en spelplan. *GameBoard* har en relation med *Player*-paketet och *Position*-paketet då den ansvarar för förflyttningen av en spelare till en ny position.

Paketet *Position* representerar en position i spelet med en logisk typ. En position skapas med hjälp av *JSON* paketet och är antingen en ”normalposition”, ”cityposition” eller ”startposition”. Varje ”cityposition” är en markörposition som har olika markörer (logiska typer), så som kossa, häst, gris, kobjällra, komocka och tom.

5.3 Relation mellan domän- och designmodell



Figur 3: Domänmodell



Figur 4: Designdiagram av modellen.

I domänmodellen har vi en komponent ”spelplan” som vi har implementerat som ett paket i projektet. Samt implementeras även komponenterna ”spelare” och ”position” i domänmodellen som paket i projektet. Det som skiljer sig från domän- och designmodellen är att det i domänmodellen finns markörer som en egen enhet, medan de i designmodellen representeras genom ett *LogicType*-fält tillhörande respektive markörposition. Komponenten ”spelare” i domänmodellen representeras som en spelpjäs i projektets vy. I domänmodellen finns komponenten ”tärning” som passar bäst som en enskild klass i projektet.

Relationerna i domänmodellen är samma som i designmodellen så när som på ett ytterligare beroende mellan spelplanen och tärningen.

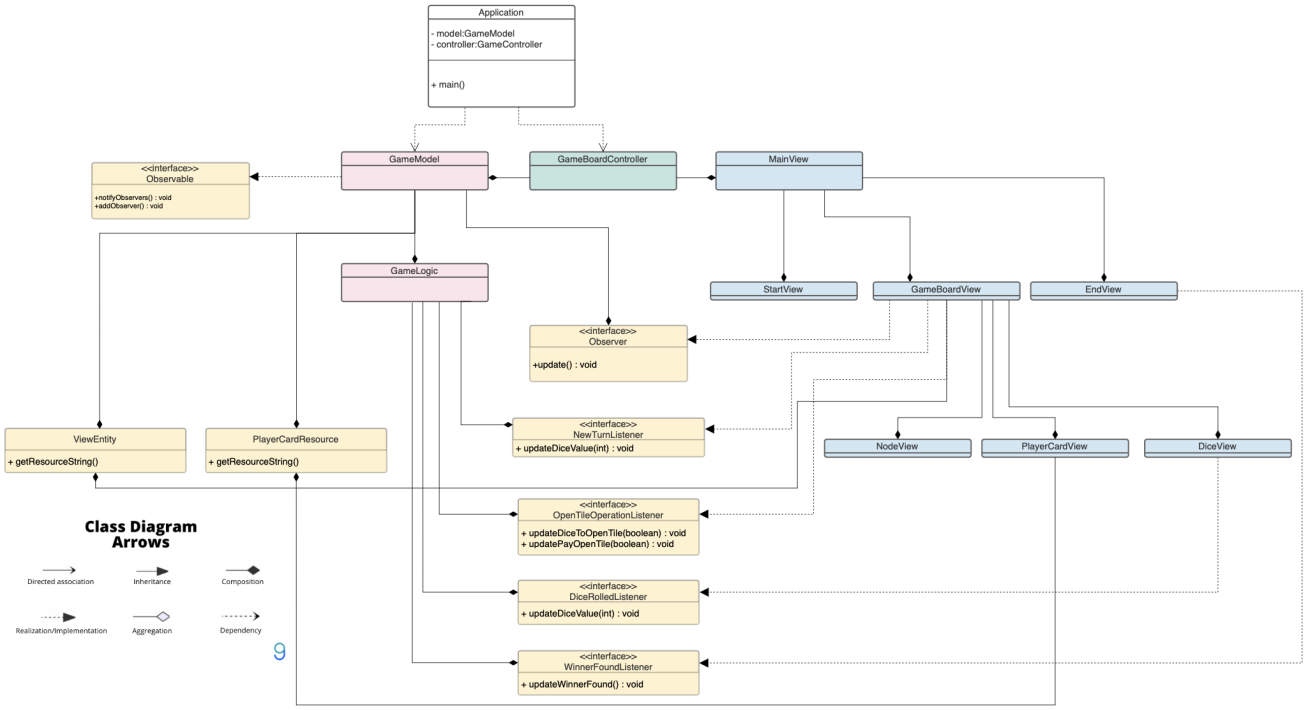
I designmodellen finns klassen *GameLogic* som inte har någon direkt representation i domänmodellen eftersom det är irrelevant i sammanhang där modellen presenteras så ytligt. I koden förvaltar denna klass all logik, samt även beroendet mellan spelplan och spelare.

Ett beroende som är konsekvent mellan de två modellerna är relationen mellan spelplan och position.

Högst upp i domänmodellen ligger *Den försvunna kossan* som representerar klassen GameModel i designmodellen och projektet. GameModel är det yttersta omslaget av modellen och är därmed det gränssnitt som kommunicerar med kontrollen.

5.4 Implementation av MVC

Applikationen implementerar designmönsret MVC (Model, View, Controller) vilket innebär att den är uppbyggd av tre delar: En modell med all logik, en vy som visar delar av modellen för användaren och sist en kontroller som tar emot input från användaren och utifrån den, utlösa förändring i modellen eller vyn. I en ideal implementation av MVC-mönstret är modellen smart, vyn dum och kontrollen tunn. För att uppnå detta har programmet delats upp i tre kataloger, en för vardera MVC-komponent. Vy-katalogen innehåller endast logik för att rita upp spelplanen utifrån modellen, samtliga hanterare av användarinput finns i kontroller-mappen, Spelplanen har tillgång till modellens spellogikmetoder så som *makeDiceRoll()* och *makePlayerMove()* för att den ska kunna påverka modellens tillstånd. För att undvika cirkelberoende implementerar modellen gränssnittet *Observable* och vyn gränssnittet *Observer*. En *Observable* har en lista med *Observers* som den efter att ha förändrats kan anropa *update()*-metoden till alla element i listan. Detta triggar bland annat spelplansvyn att rita om spelplanen på nytt för att visa programmets nya tillstånd. Delar av vyn implementerar även olika typer av *Listeners* som modellen meddelar när något har skett. Exempelvis implementerar klassen *DiceView* interface:t *DiceRolledListener* och blir meddelad av modellen när tärningen har slagits. Modellen har ingen vetskap om vem eller vilka som lyssnar och vyn vet inget mer än att den ska uppdateras. Paketet Utils abstraherar modellen för vyn.



Figur 5: Designdiagram av MVC implementation. Modellen representeras av av de rosamarkerade klasserna. Vyn representeras av de blåmarkerade klasserna. Kontrollen representeras av den grönmarkerade klassen. Paketet *Utils* representeras av de gulmarkerade klasserna.

5.5 Designmönster

5.5.1 Factory Pattern

Designmönstret *Factory Pattern* används ett flertal gånger i koden, exempelvis för att skapa spelare (med *PlayerFactory*) och positioner (med *PositionFactory*). Den främsta anledningen till varför designmönstret har valts att användas är för att klasser inte skall behöva veta om instansieringen av vissa objekt. Exempelvis ska inte klassen *GameLogic* veta eller ansvara för hur spelare skapas, utan kan istället använder sig av *PlayerFactory* för att initiera listan med spelare.

5.5.2 Observer Pattern

Programmet använder sig av *Observer Pattern* då vyn uppdateras efter att modellen kallat på sina observatörer att uppdateras, vilket beskrivs ovan (5.4).

5.5.3 Singleton Pattern

Programmet använder sig även av *Singleton Pattern* i klassen GameLogic. *Singleton Pattern* möjliggör en kontrollerad behörighet att påverka spelets komponenter, eftersom det endast kan finnas en instans som utövar logikanrop.

5.5.4 Interface Segregation

Programmet bygger på principen *Interface Segregation*. För varje nytt behov av ett interface, har ett nytt skapats snarare än att ett redan existerande breddats.

6 Kvalitet

6.1 Åtkomstkontroll och säkerhet

NA/IA

6.2 Test

I projektet används JUnit för att genomföra enhetstestning. Tester ligger i mappen ”Test”. Samtliga klasser i modellen testas för att säkerställa korrekt funktionalitet. Modellen och tillhörande Utils testas utförligt och når 95 procent radtäckning. Testning av vyn och kontrollern är överflödigt på grund av projektets storlek. Dessa klasser är relativt få och kan inte genomföra förändringar på något objekt utan att använda modellens metoder. Testning av modell och kontroller innebär att modellen testas en ytterligare gång.

6.3 Kända problem

De problem vi känner till är:

- Knapparna håller vyn koll på, det är inte klassen *GameLogic* som bestämmer vilken vy som ska visas.

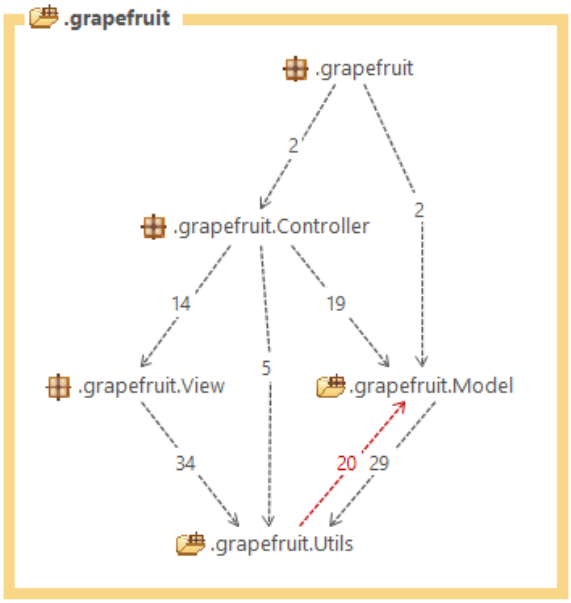
- Interface:t *IPlayer* är har för många metoder. Den borde extraheras till flera mindre.

- Skicka in information från *Choicebox*, hade varit bättre att använda sig utav *EventHandlers*. Så att eventet sker när användaren väljer antalet spelare från *Choicebox:n*.

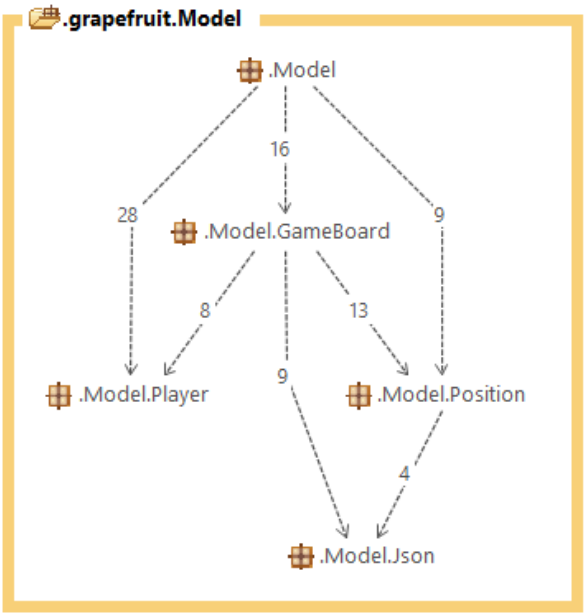
- Metoden *redrawChildren()* tar bort samtliga noder från *background.fxml* och ritar ut dem på nytt. Det hade varit bättre att ändra de komponenter som har förändrats med hjälp av CSS istället för att använda flera olika FXML-filer.

- I klassen *PlayerCardView* jämförs nuvarande spelares id med klassen *PlayerCardResource* spelares id. Detta är för komplicerat för vyn att hantera, då man ska följa den fundamentala idén med MVC.

6.4 Beroendediagram



Figur 6: Beroendediagram av STAN



Figur 7: Beroendediagram av STAN

7 Peer-review of Grapefruit

7.1 Model

GameModel:

- Både observers och players är package-private, varför? GameBoard är public, farligt då vem som helst har tillgång till referensvariabeln.

GameBoard:

- Alla globala variabler är package-private, avsiktligt? Om inte borde dessa sättas till private.
- movePlayer-metoden overloadas med två olika input-parametrar, men enligt medföljande JavaDoc ska metoderna göra två skilda saker. En av metoderna borde byta namn.
- Metoden rollDice returnerar för närvarande ett mock-värde 2, den senare implementerade logiken bör extraheras till en Dice klass som har en liknande metod, precis som ni har tänkt i er UML.

Map:

- Metoden deHighlight ”släcker” samtliga noder, detta borde eventuellt reflekteras i namnet på metoden.

Node:

- Metoden addRelatedNode har följande kod: if (relatedNodes == null) relatedNodes = new ArrayList<>(); relatedNodes.add(node);

  Lättare vore att bara initiera listan direkt uppe vid konstruktorn som ej har listan som parameter:

  public Node(IPosition position) this.position = position; relatedNodes = new ArrayList<>();

MapFactory:

- Klassen har en publik implicit konstruktor, så oavsett att den har en statisk initiations-metod kan man initiera den med en tom konstruktor, vilket inte är intentionen och borde förhindras.

PlayerFactory:

- Klassen har en publik implicit konstruktor, så oavsett att den har en statisk initiations-metod kan man initiera den med en tom konstruktor, vilket inte är intentionen och borde förhindras.
- Returnerar null om spelarantalet som angetts är över 4, farlig kod generellt. Antalet skulle kunna kontrolleras innan listan skapas, den borde definitivt inte

vara null. Initieringenmetoden skulle också kunna kasta ett undantag om metodförhållandena inte uppfylls.

PlayerColor:

- JavaDoc för metoden evaluateResourceString lyder ”Evaluate the player color and match it with the right view-resource”, då låter det som att denna resurs borde ligga i view-paketet, inte i modellen.

NormalPosition:

- Här är också visuella aspekter av programmet, i modellen:

  @Override public String getResourceString() if (isHighlighted) return node-view-highlighted.fxml"; return node-view.fxml";

  Dessa borde också flyttas till view.

7.2 View

GameBoardView:

- Alla globala variabler är package-private, avsiktligt eller kan de eventuellt göras private? För nuvarande kan de det då paketet endast innefattar NodeView, som inte utnyttjar tillgången.
- Behöver den ha en public konstruktor? För tillfället kan man instansiera klassen direkt från den publika konstruktorn.
- GameBoardView är FXMLLoader klassens kontroller, känns inte helt intuitivt?
- Metoden redrawChildren skapar nya FXMLLoaders för varje positionerbart objekt när update- metoden kallas, förmodligen inte så processoreffektivt?
- Dessutom använder den en ”enhanced for-loop”, men använder en yttre variabel i för att inkrementera och se efter när loopandet är färdigt. Känns som att det skulle vara snyggare med en konventionell loop.

NodeView:

- Variabelnamnen kan vara mer genomtänkta (e.g. x,y; i förhållande till vad?, position som refererar till en FXML-cirkel).
- Återkomstmodifierare används blandat (e.g position är public, varför?)

7.3 Kontroller

GameBoardController:

- GameBoardView initieras här, borde försökas initieras i HelloApplication istället. Om inte view kan initieras utan kontroller känns reparationen mellan de två onödig då de ändå har ett starkt beroende av varandra.

7.4 Övrigt

Koden är i stora drag byggt efter MVC-mönstret och uppfyller det i stor utsträckning, med de in- bakade visuella aspekterna som undantag.

Designprinciper följs och en reparation av koncern mellan klasser görs. Ett exempel är implementring av designmönstret Observer; förövrigt används Interface Segregation flitigt. Dessutom använder flera klasser moönstret Factory.

Koden är i största laget konsekvent, med såväl konventioner som intuitiv namngivning på paket, klasser, metoder och variabler.

På grund av stadig implementing av gränssnitt är koden betydligt lättare att återanvända då klienten är mer begränsad i vilka användningsområden koden kan nyttjas.

Eftersom klassuppdelningen är gedigen blir underhållningen av koden betydligt lättare. Funktionalitet kan enkelt läggas till / tas bort vilket är tacksamt så man slipper utföra shotgun surgery.

JavaDoc finns på flertalet metoder men är lite inkonsekvent på andra. Ett behov av klassbeskrivningar finns på samtliga klasser. En mindre dokumenteringsinsats skulle eliminera problemet.

Koden är för tillfället väldigt bergränsat testad och kommer i framtiden behöva utöka testarean, vilket gruppen själva specificerar i sin definition av färdig-sektion.

Det finns vissa säkerhetsproblem med offentliga åtkomstmodifierare som omotiverat definierats så. Detta borde definitivt ändras.

Den kontinuerliga initieringen av nya FXMLLoader-objekt skulle på större skala kunna orsaka pre- standaproblem och borde överses för en bättre lösning.

Koden är väldigt enkel att förstå och tydligt strukturerad på en syntax-nivå. Ett utförligt UML- diagram hade underlättat analysen.

8 Peer review of ESSBG by Grapefruit

The project uses a consistent coding style. Apart from some unnecessary blank lines, we have no other remarks on the code style.

The code uses several design patterns which makes the project object oriented. For example the project uses Factory pattern, for creating Monuments and different amounts of Cards. The Observer pattern is used when Render listens to the Network. Also, Template pattern, Decorator pattern and Adapter pattern is used.

The code is pretty well documented. JavaDocs need some improvements, describe the parameters and return parts. Some files are well documented and some are not, so it's a bit uneven. You could add @author at several classes.

We would say that the project has proper names. Good and describing names.

After an overview of the code we couldn't find any unnecessary or circle dependencies.

We could see that you tested a lot of the model package, but you could test a bit more to cover more lines of code.

Since we don't have much experience of working with servers, we struggle a bit with understanding how it all works. But looking at individual classes and files we understand their functionality.

The code in regards to MVC is separated into individual directories which gives a clear overview of MVC. Also, the Model is independent which means that we can easily add more functionality to the Model without messing up the view. The view can also be updated without messing up the Model.

After looking at the View, we found that the view seems to have the responsibilities that a view should have (only graphical methods). Though we noticed that the "show" methods in GameScene and LobbyScene contain a lot of code which could be separated into smaller ones. On the other hand, we think you did a good job splitting up methods in the class DrawableBoard.

The code for Card and Monument is reusable and extendable due to their abstract classes and use of enums and factories.

There are a lot of classes which depend on the org.json library. This could be improved by implementing some kind of jsonHandler that contains the json functionality throughout the program. Regarding performance issues, we noticed that Mac users are not able to run the project via intellij. Moreover, we couldn't figure out how to actually start the game.

There are some classes with a lot of instance variables and sometimes there is a purpose for that. Overall, it is a great start to your project. Both SDD and RAD are adequate

and include good descriptions of the project. More JavaDocs and tests are some things to consider. Keep up the good work!

9 Referenser

JUnit, 2021. Hämtad från https://junit.org/junit5/.

Travis, 2021. Hämtad från https://www.travis-ci.com/.

Maven, 2021. Hämtad från https://maven.apache.org/install.html.

GSON, 2021. Hämtad från https://github.com/google/gson.

JSON, 2021. Hämtade från https://www.json.org/json-en.html.