Designdokument för mjukvaran till 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 placerade runt om på spelplanen, och sedan ta sig i mål. Spelarna turas om att slå en tärning och förflytta sin spelpjäs så många steg som tärningen visar. När en spelare 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, beroende på vilken kommer spelaren att påverkas på olika sätt.

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, så väl som en dator.
- GSON - Är ett Java-bibliotek som bland annat kan användas till att konvertera en JSON sträng till motsvarande Javaobjekt.
- JUnit - Är ett ramverk att använda vid enhetstestning i Java.
- Modell - utgör logiken för den information som applikationen bearbetar.
- Vyn - Exponerar delar av modellen för användaren.
- Kontroller - Tar emot input från användaren för att utlösa förändring i vyn eller modellen.

2 Systemarkitektur

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.

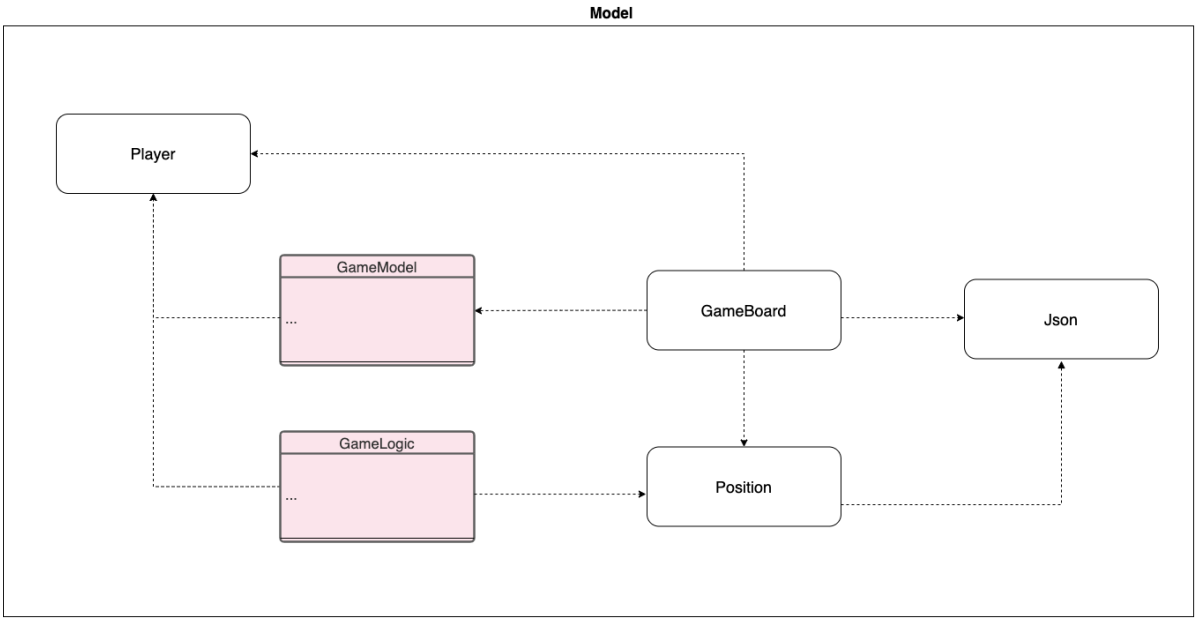
När modellen initierats från kontrollen skapas en spelplan utifrån en JSON-fil, samt rätt antal spelare. Detta sker genom klassen *JSONHandler* som använder sig av GSON för att konvertera JSON-filen till ett javaobjekt, i detta fall *JSONBoardReader*.

3 Systemdesign

3.1 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.

3.2 Relation mellan paket



Figur 1: Designdiagram av kopplingen mellan paketen i modellen.

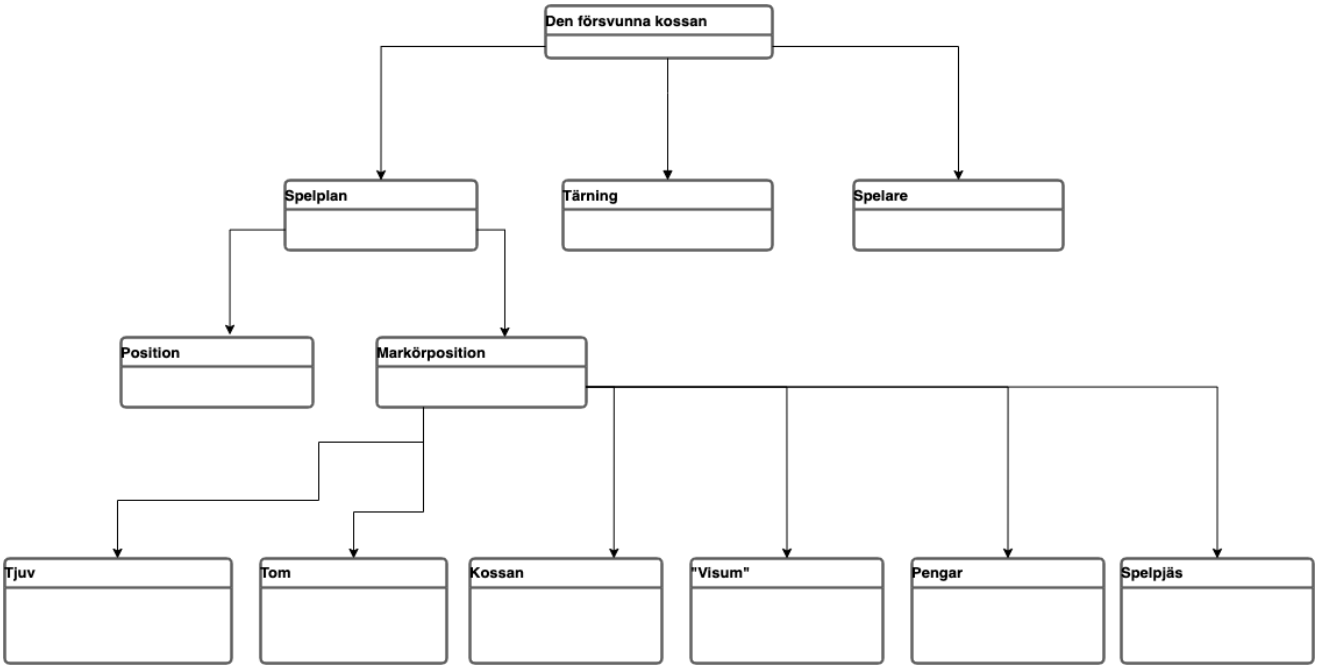
Paketet *Model* består av fyra paket som har ansvar för all logik för spelet, där vardera paket representerar en komponent i applikationen.

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.

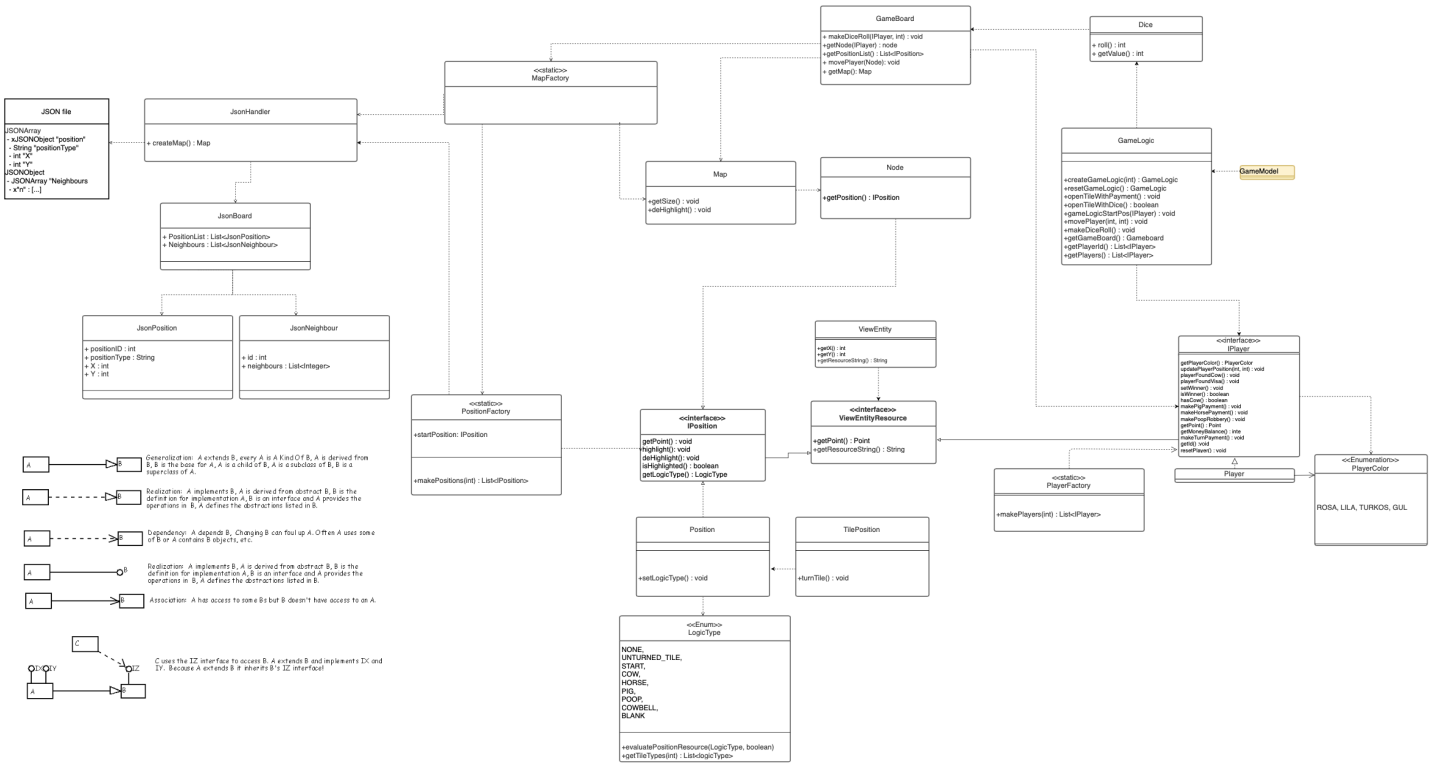
Paketet *GameBoard* representerar en spelplan i spelet. *GameBoard* använder paketet *JSON* för att skapa en spelplan. *GameBoard* har en relation med *Player*-paketet och *Position*-paketet då den ansvarar för förflyttningen av en spelare till en ny position.

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.

3.3 Relation mellan domän- och designmodell



Figur 2: Domänmodell



Figur 3: 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.

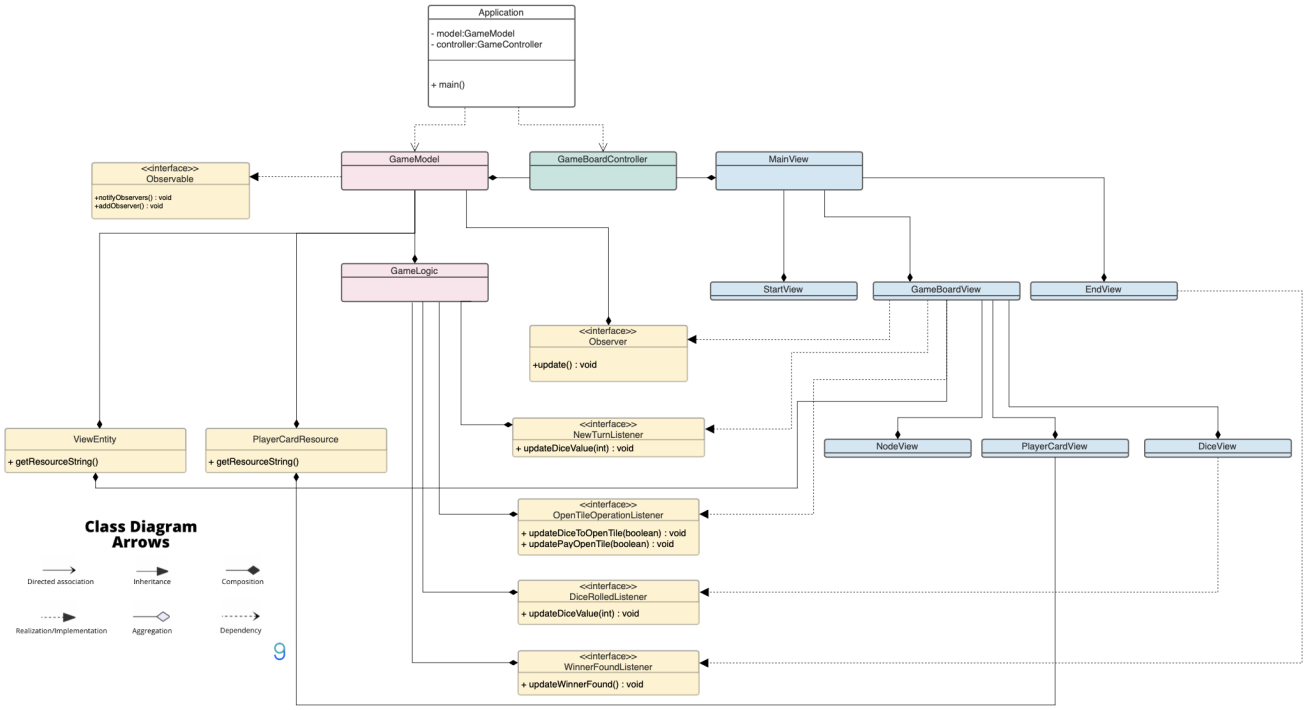
I designmodellen finns klassen *GameLogic* som inte har någon direkt representation i domänmodellen eftersom det är irrelevant i sammanhang där modellen presenteras så ytligt. I koden förvaltar denna klass all logik, samt även beroendet mellan spelplan och spelare.

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.

3.4 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 4: 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.

3.5 Designmönster

3.5.1 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.

3.5.2 Observer Pattern

Programmet använder sig av *Observer Pattern* då vyn uppdateras efter att modellen kallat på sina observatörer att uppdateras, vilket beskrivs ovan (3.4).

3.5.3 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.

3.5.4 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.

4 Ihållande datahantering

IA/NA

5 Kvalitet

5.1 Åtkomstkontroll och säkerhet

NA/IA

5.2 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.

5.3 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.

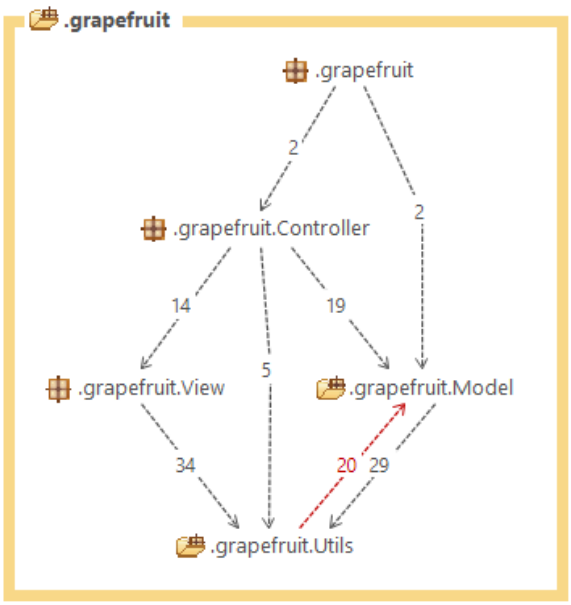
- Interface:t *IPlayer* är har för många metoder. Den borde extraheras till flera mindre.

- 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*.

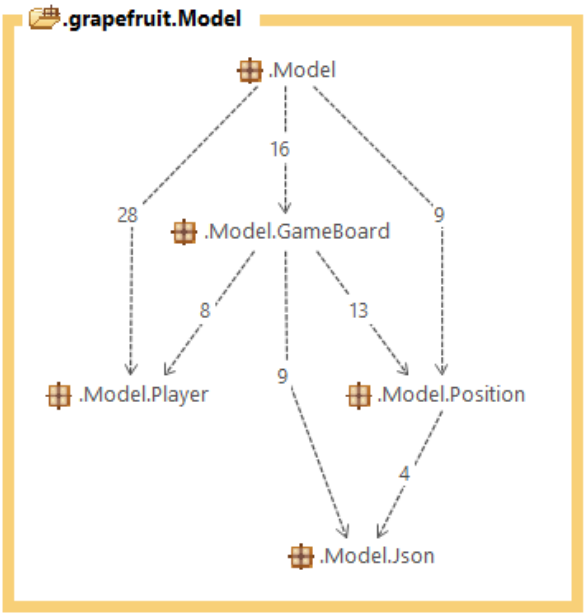
- Metoden *redrawChildren()* tar bort samtliga noder från *background.fxml* och ritar ut dem på nytt. Det hade varit bättre att ändra de komponenter som har förändrats med hjälp av CSS istället för att använda flera olika FXML-filer.

- 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.

5.4 Beroendediagram



Figur 5: Beroendediagram av STAN



Figur 6: Beroendediagram av STAN

6 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.