Requirements and Analysis Document for Hive

Hanna Adenholm, Lisa Qwinth, Stina Hansson

2021-10-24

Version 1

1. Introduktion

Hive är ett brädspel för två spelare där hexagon-formade pjäser som föreställer insekter utgör brädet. Målet med detta projekt är att omvandla brädspelet Hive till en mobilapplikation, och därigenom göra det mer tillgängligt för folk att spela. Spelet som har tagits fram består av fem olika pjäser, biet, gräshoppan, myran, spindeln och skalbaggen. Dessa pjäser rör sig på olika sätt på brädet. Biet kan, likt kungen i schack, endast gå ett steg i valfri riktning. Gräshoppan kan hoppa över brädet, men endast i raka linjer. Myran kan gå hur många steg som helst runt brädet. Spindeln är som myran, men kan endast gå tre steg. Skalbaggen är som biet, men kan också hoppa upp på andra pjäser. Målet med spelet är att med hjälp av pjäserna omringa sin motståndares bi, men samtidigt se upp så att ens eget bi inte blir omringat. Spelet är gjort för att spelas på samma mobil, där spelarna får turas om att göra sitt drag.

Alla regler: https://www.ultraboardgames.com/hive/game-rules.php

Applikationen Hive är ett alternativ till de som föredrar att spela mobilspel framför brädspel, eller för de som inte vill lägga pengar eller har råd med ett brädspel. Applikationen har fördelar som det fysiska brädspelet inte har i och med att man alltid har tillgång till det via mobilen och man inte är beroende av en slät yta att lägga ut pjäserna på.

1.1 Definitioner

- **Activity:** En skärm som användaren kan interagera med där gränssnittet visas
- **GUI:** Grafiskt användargränssnitt
- **Hashmap:** En lista av par, där paret består av en nyckel och ett värde
- **Enumeration:** En sorts klass i java som håller en grupp av konstanter, vilket är variabler som inte kommer förändras.
- **Fragment:** En del av gränssnittet som läggs till på en activity
- **Java:** Ett objektorienterat programspråk
- **JUnit:** Ett ramverk för enhetstestning, för Java
- **LiveData:** En klass som finns i Androids bibliotek som notifierar de klasser som observerar viss data så fort denna data ändras
- **User Story:** En förklaring av en funktion i en applikation, som är skriven från perspektivet av en användare.

2. Krav

2.1 User Stories

De user stories som projektet är byggt på listas nedan. Alla user stories är implementerade förutom STK006.

User Story

ID : STK001
Namn: Rutnät med pjäser

Beskrivning

Som användare vill jag kunna se rutnätet med de spelade pjäserna så att jag kan spela spelet.

Bekräftelse

Funktionell

- Kan jag se rutnätet med de spelade pjäserna?
- Kan jag se alla pjäser på en gång?

User Story

ID : STK002
Namn: Möjliga drag

Beskrivning

Som användare vill jag kunna se möjliga drag för en specifik pjäs så att jag kan veta var den kan flyttas.

Bekräftelse

Funktionell

- Kan jag klicka på en spelad pjäs och se de möjliga dragen markerade på rutnätet?
- Är den valda pjäsen markerad?
- Kan jag ändra mig och klicka på en annan pjäs?

User Story

ID : STK003
Namn: Ospelade pjäser

Beskrivning

Som användare vill jag kunna se de ospelade pjäserna, så att jag kan veta hur jag ska göra mitt drag.

Bekräftelse

Funktionell

- Kan jag se alla pjäser som inte är spelade än?
- Kan jag se hur många som är kvar av varje typ?

User Story

ID : STK004
Namn: Spela en ny pjäs

Beskrivning

Som användare vill jag kunna lägga ut en ny pjäs, så att jag kan spela spelet.

Bekräftelse

Funktionell

- Kan jag klicka på en pjäs i min hand och se de möjliga dragen markerade?
- Kan jag klicka på en av de möjliga dragen och se att min pjäs flyttas dit?

User Story

ID : STK005
Namn: Nuvarande spelare

Beskrivning

Som användare vill jag kunna se vems tur det är, så att jag vet när det är min tur.

Bekräftelse

Funktionell

- Ändras färgtemat på skärmen beroende på vems tur det är?
- Byts spelarhanden till den nuvarande spelarens?
- Står det vems tur det är överst på skärmen?

User Story

ID : STK006
Namn: Regelfönster

Beskrivning

Som användare vill jag ha en knapp som visar reglerna för spelet så att det blir enklare att lära sig spelet.

Bekräftelse

Funktionell

- Finns det en knapp lättillgänglig med en symbol som förmedlar att den visar reglerna?
- Kommer reglerna upp när man tryckt på knappen?

User Story

ID : STK007
Namn: Starta om spelet

Beskrivning

Som användare vill jag ha en knapp som startar om spelet så att jag kan starta ett nytt spel när jag vill.

Bekräftelse

Funktionell

- Finns det en knapp lättillgänglig med en symbol som förmedlar att den startar om spelet?
- Ligger knappen så man inte kan komma åt den av misstagen medan man spelar?
- Startas spelet om när jag klickar på knappen?

User Story

ID : STK008
Namn: Spelregler

Beskrivning

Som användare vill jag att en pjäs inte flyttas från en position till en annan om den inte får det enligt spelreglerna

Bekräftelse

Funktionell

- Är endast tillåtna drag markerade som möjliga drag när jag klickar på en pjäs?

User Story

ID : STK009
Namn: Ny skärm vid vinst

Beskrivning

Som användare vill jag se en ny skärm med vilken spelare som vann när någons drottning är omringad, för att veta när spelet är över och vad resultatet blev.

Bekräftelse

Funktionell

- Kommer det upp en ny skärm när spelet är över?
- Visar skärmen vem det är som vunnit?

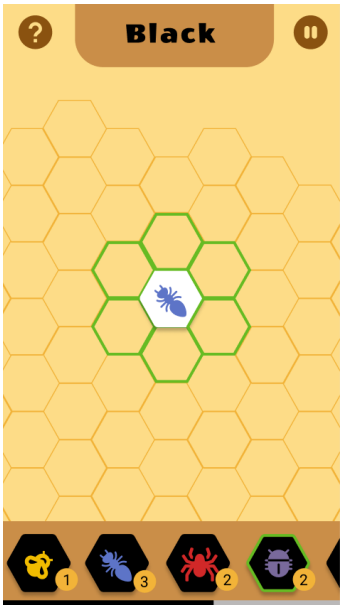
2.2 Definition of Done

För att säkerställa att hela gruppen är överens om när en user story anses som klar, har gruppen satt upp en Definition of Done-lista.

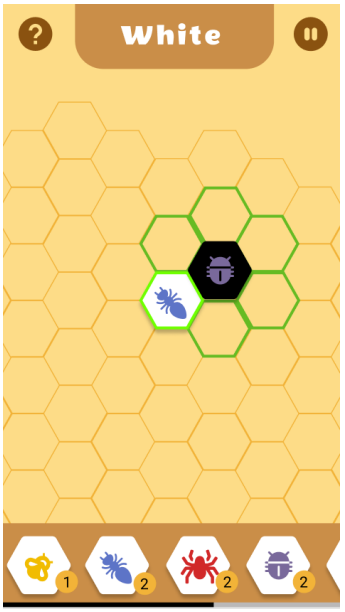
- Koden ska genomföra det den förväntas göra, utan komplikationer
- Koden ska vara bra ur ett objektorienterat perspektiv
- Kraven för den aktuella User Storyn ska vara mött
- Test som testar aktuella metoder ska finnas och bli godkända
- All kod ska vara dokumenterad

2.3 Användargränssnitt

I figur 1 och 2 syns prototypen som gjordes i Figma där man kan se hur GUI:et hade sett ut om applikationen utvecklats vidare och funktionalitet såsom en hjälp-knapp med regler hade implementerats.

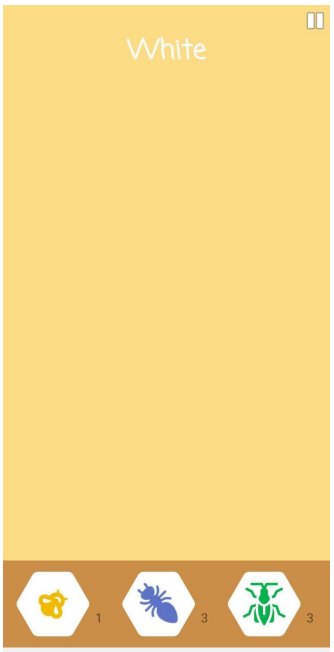


Figur 1: Figmaprototyp för ett drag

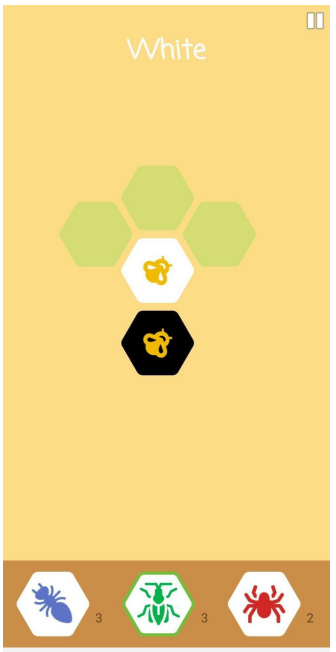


Figur 2: Figmaprototyp för ett drag

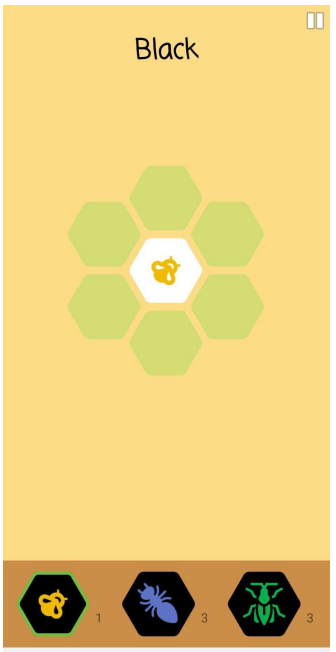
När applikationen sätts igång kommer man till huvudsidan (se figur 3). Ett nytt spel skapas och startas automatiskt. Genom att klicka på pjäserna och sedan på de upplysta platserna kan man flytta och placera ut pjäser. Vyn för hur det ser ut när den vita respektive svarta spelaren ska göra ett drag kan ses i figur 4 och 5.



Figur 3: Huvudsida



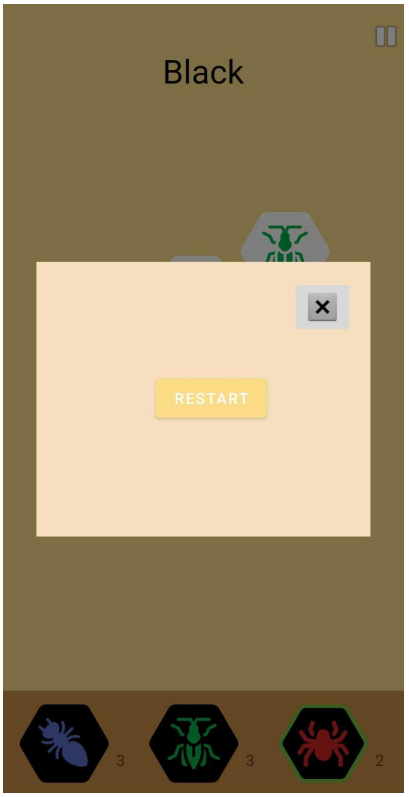
Figur 4: Vy vid vita spelarens tur



Figur 5: Vy vid svarta spelarens tur

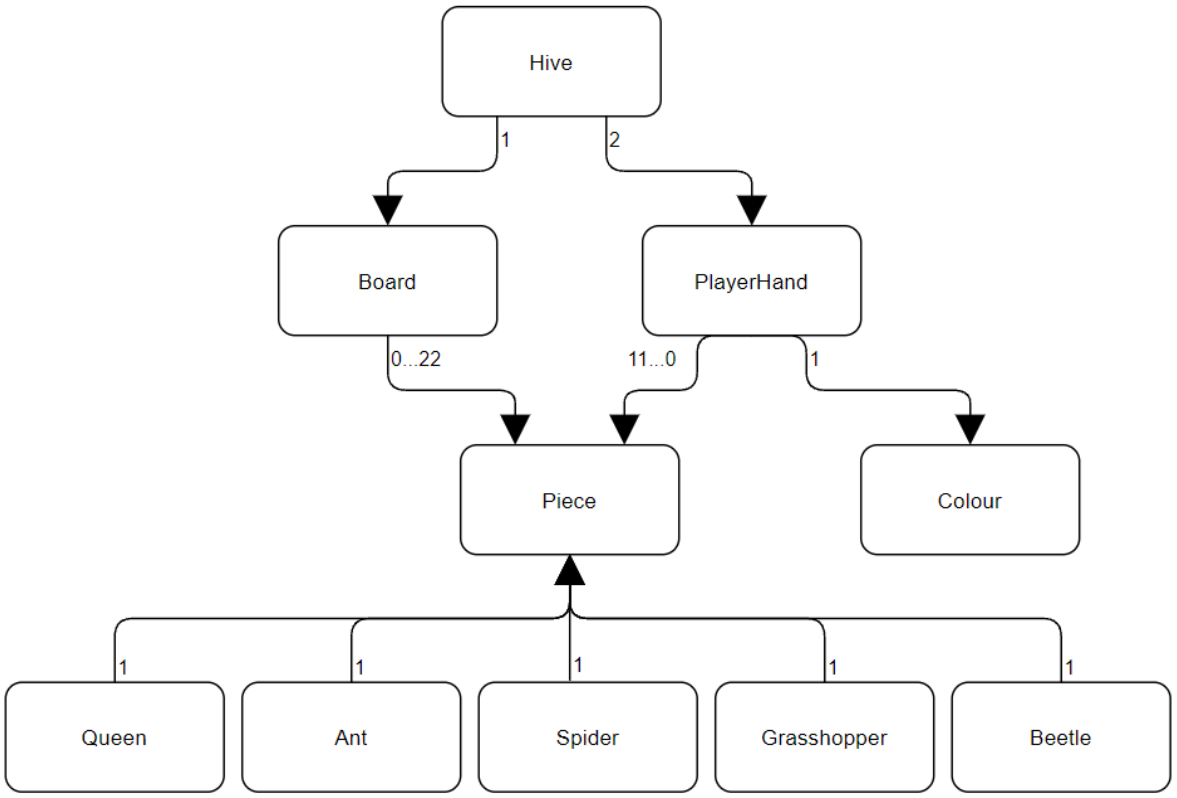
När en spelare vinner dyker en ruta upp (se figur 6) som meddelar vem som vunnit, och ger möjlighet att starta ett nytt spel. Högst upp i högra hörnet finns en pausknapp som tar fram en ruta (se figur 7) där man kan välja att starta om spelet.

Figur 6: Vinstruta



Figur 7: Pausmeny

3. Domänmodell



3.1 Beskrivning av klasserna

Hive
Representerar spelet. Innehåller en board och två playerHands och håller koll på vems tur det är, vilken piece som är vald och om det är någon piece vald.

Board
Håller koll på pjäserna och deras position på brädet, där brädet representeras av en hashmap. Brädet har en variabel för varje drottning och håller koll på när en drottning blivit omringad och spelet då är slut.

PlayerHand
Har koll på vilka pjäser som befinner sig i respektive spelares hand. När ett objekt av PlayerHand skapas, skapas pjäserna som varje spelare ska ha från början.

Colour
En Enumeration som håller de två färgerna som en spelare kan ha, svart och vit.

Piece
I Piece finns de regler för hur pjäserna rör sig som är gemensamma för alla pjäser.

Queen
Ansvarar för hur drottningen kan röra sig. Drottningen kan enbart röra sig ett steg i valfri riktning, så länge det inte redan ligger en pjäs där.

Ant
Ansvarar för hur myran kan röra sig. Myran kan röra sig valfritt antal steg i valfri riktning längs med de utlagda pjäserna.

Grasshopper
Ansvarar för hur gräshoppan kan röra sig. Gräshoppan kan hoppa över andra pjäser i raka linjer, men inte över tomma platser.

Beetle
Ansvarar för hur skalbaggen kan röra sig. Skalbaggen kan röra sig ett steg i valfri riktning, och har förmågan att krypa upp på andra pjäser.

Spider
Ansvarar för hur spindeln kan röra sig. Spindeln kan röra sig exakt tre steg längst med de utlagda pjäserna, i valfri riktning.

4. Referenser

6.1 Verktyg

- **Android Studio** IDE:n vi gjorde projektet med.
- **Draw.io** Ett verkyg som användes för att måla UML-diagram.
- **Figma** Ett verktyg som användes för att ta fram den grafiska designen.
- **GitHub** Lagring och versionshantering för Git av projektet.
- **Google Drive** Samlad lagring av samtliga dokument rörande projektet.
- **Slack** Kommunikationstjänst som användes för kommunikation om projektet
- **Trello** Organiseringsverktyg för de User Storys vi jobbat utifrån i projektet
- **Zoom** Kommunikationsverktyg för videosamtal för samtal om projektet.

6.2 Bibliotek

- **JUnit** används för att testa programmet.

System design document for Koworkers's Hive

Adenholm Hanna, Hansson Stina, Qwinth Lisa
2021-10-24
version 1

1. Introduktion

Hive är ett brädspel för två spelare där hexagon-formade pjäser som föreställer insekter utgör brädet. Målet med detta projekt är att omvandla brädspelet Hive till en mobilapplikation, och därigenom göra det mer tillgängligt för folk att spela. Spelet består i nuläget av fem olika pjäser, biet, gräshoppan, myran, spindeln och skalbaggen. Dessa pjäser rör sig på olika sätt på brädet. Biet kan, likt kungen i schack, endast gå ett steg i valfri riktning. Gräshoppan kan hoppa över brädet, men endast i raka linjer. Myran kan gå hur många steg som helst runt brädet. Spindeln är som myran, men kan endast gå tre steg. Skalbaggen är som biet, men kan också hoppa upp på andra pjäser. Målet med spelet är att med hjälp av pjäserna omringa sin motståndares bi, men samtidigt se upp så att ens eget bi inte blir omringat.

I detta dokument kommer applikationens systemarkitektur och systemdesign diskuteras, samt kvaliteten på projektet.

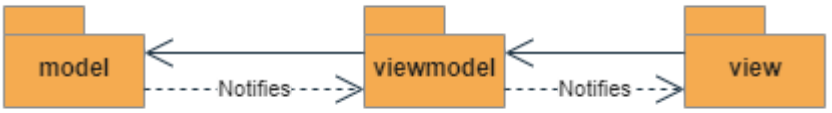
1.1 Definitioner

- **Activity:** En skärm som användaren kan interagera med där gränssnittet visas
- **GUI:** Grafiskt användargränssnitt
- **Hashmap:** En lista av par, där paret består av en nyckel och ett värde
- **Enumeration:** En sorts klass i java som håller en grupp av konstanter, vilket är variabler som inte kommer förändras.
- **Fragment:** En del av gränssnittet som läggs till på en activity
- **Java:** Ett objektorienterat programspråk
- **JUnit:** Ett ramverk för enhetstestning, för Java
- **LiveData:** En klass som finns i Androids bibliotek som notifierar de klasser som observerar viss data så fort denna data ändras
- **Observer** Strukturellt designmönster, där ett objekt håller en lista av observatorer som den kan notifiera för att uppdatera
- **Singleton** Designmönster där det endast skapas en instans av ett objekt.
- **User Story:** En förklaring av en funktion i en applikation, som är skriven från perspektivet av en användare.

2. Systemarkitektur

Applikationen Hive är i nuläget inte uppkopplad till internet, och använder inga servrar eller databaser. Detta gör att applikationen inte har någon komplicerad systemarkitektur.

3. Systemdesign



Figur 1: Package-UML

Till den grundläggande programstrukturen användes designmönstret MVVM (Model, View, ViewModel) vilket är strukturerat så att programlogiken och gränssnittet implementeras separat från varandra. Att hålla logiken och kontrollerna för gränsnittet separerat från varandra gör att koden mycket lättare kan återanvändas och blir lättare att förstå sig på, utveckla och underhålla. Se figur 1.

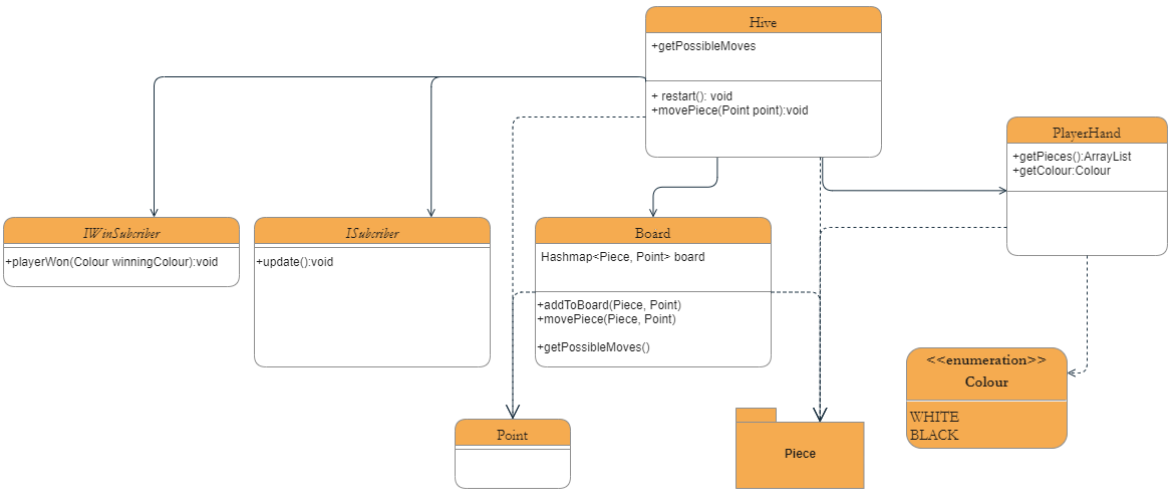
I MVVM- mönstret så fungerar vy-modellen som en mellanhand mellan vyn och modellen och det är vy-modellens jobb att hantera de klickningar som vyn registrerar, och att presentera datan som vyn sen ska visa. För att vy-modellen ska vara livscykeloberoende och kunna kopplas till olika vyer så får vy-modellen inte ha något beroende på vyn. Detta behövs till exempel vid skärmrotation då den aktiva aktiviteten stängs ner och en ny skapas.

I och med att programmet skapats med android finns det bra verktyg för att kunna binda datan i vy-modellen till vyn. Ett bra exempel på detta är LiveData. Detta används i vy-modellerna som finns för board och playerhand. Med LiveData så notifieras alla vyer som observerar denna data så fort datan ändras.

Modellen ansvarar för att ta hand om all spellogik och innehåller all data. För att denna lätt ska kunna återanvändas till andra system/program så har beroenden till andra bibliotek och dylikt minimerats. Till exempel har en egen Point-klass skapats istället för att importera en från till exempel JavaFx eller androids bibliotek. se figur 2.

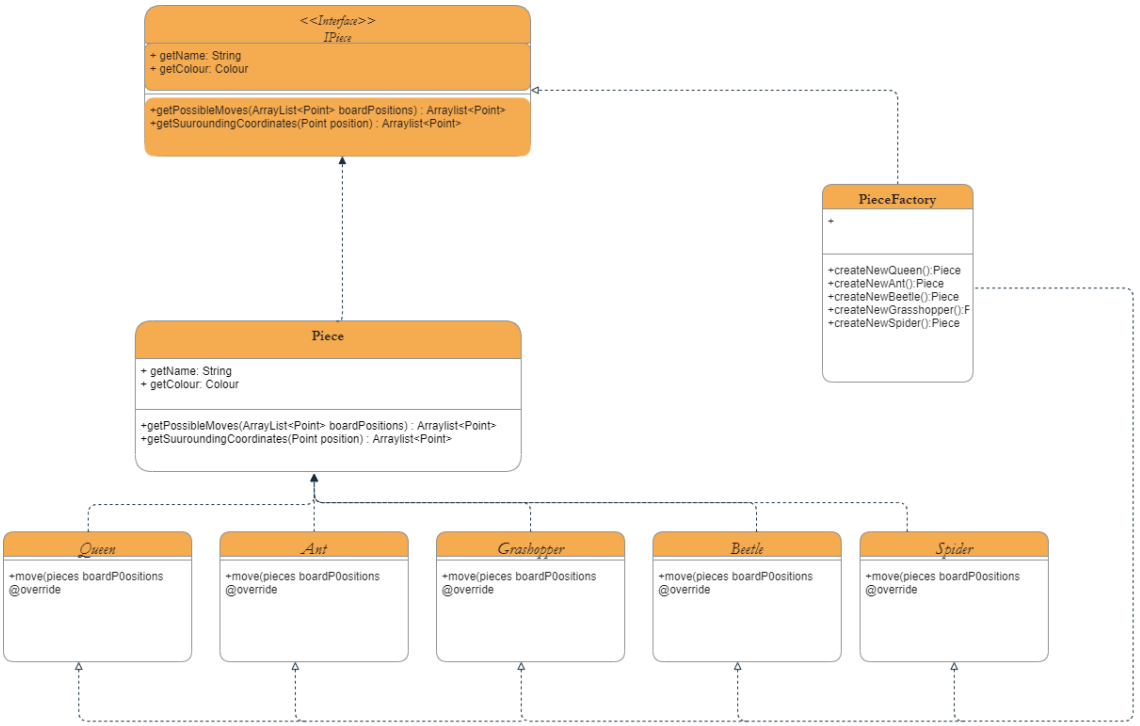
Modellen har inte heller några beroenden på vyn eller vy-modellen. Men för att vy-modellen ska få reda på när något i modellen har uppdaterats har designmönstret observer använts. Detta innebär att modellen innehåller en lista med subscribers som implementerar interfacet Isubscriber och när den uppdateras så kallar den på updateringsmetoden i alla subscribers som ligger i listan. Det är då vy-modellerna som implementerar Isubscriber-interfacet.

Från början implementerade Hive-klassen, som är den klass som representerar spelet, singleton-mönstret. Detta då det gjorde det väldigt lätt att komma åt instansen av hive från de olika vy-modellerna och att man då inte behövde komma åt alla vymodeller från huvudaktiviteten för att lägga till samma hiveinstans i dem. Detta finns dock inte kvar då singleton bryter mot single responsibility principle och den gör det väldigt lätt att man av misstag kan komma åt och ändra i den enda instansen av Hive som finns.



Figur 2: UML-diagram av model-Package

För att modellen lätt ska kunna utvecklas och för att man lätt ska kunna lägga till fler sorters pjäser så implementeras designmönstret factory. Detta har då gjorts genom att alla pieces implementerar interfacet IPiece och att factory metoderna som finns i factory-klassen skapar de enskilda pieces och sen returnerar dem som en IPiece. Detta gör att kunden inte behöver veta vilken speciell implementering av IPiece piecen är utan den kan behandla alla pieces likadant vilket då gör det lätt om man vill lägga till andra sorters pieces. Att Piecefactory ligger som en egen klass gör också att de enskilda piece-klasserna kan vara package-private vilket kan förhindra onödiga beroenden. se figur 3.



Figur 3: UML-diagram av piece-package

4. Persistent data management

N/A

5. Kvalitet

5.1 Test

Applikationen testas med hjälp av JUnit, och kan hittas i test-mappen där testerna sedan är indelade efter klass eller package. Testerna täcker 99% av modellen, där de publika metoderna testas direkt, och då testas även de privata metoderna när de publika metoderna kallar på dem.

5.2 Kända problem

Det finns ett fåtal kända problem som vi av olika anledningar inte löst än. Ett av dessa är att klassen Board är relativt omfattande, och kan ses som att den bryter mot Single Responsibility Principle, då den både håller pjäsernas positioner, flyttar dem och innehåller spellogik. Det hade gått att dela upp Board, men det hade medfört fler beroenden, och enligt oss inte förbättrat koden särskilt mycket.

Det finns en bugg i hur vissa av pjäserna rör sig som ännu inte lösts, på grund av tidsbrist. I vissa specialfall kan en del av pjäserna gå till en plats den inte borde kunna, men detta sker bara när brädet ser ut på ett specifikt sätt och förstör inte spelet, vilket är varför vi inte valt att prioritera detta problem.

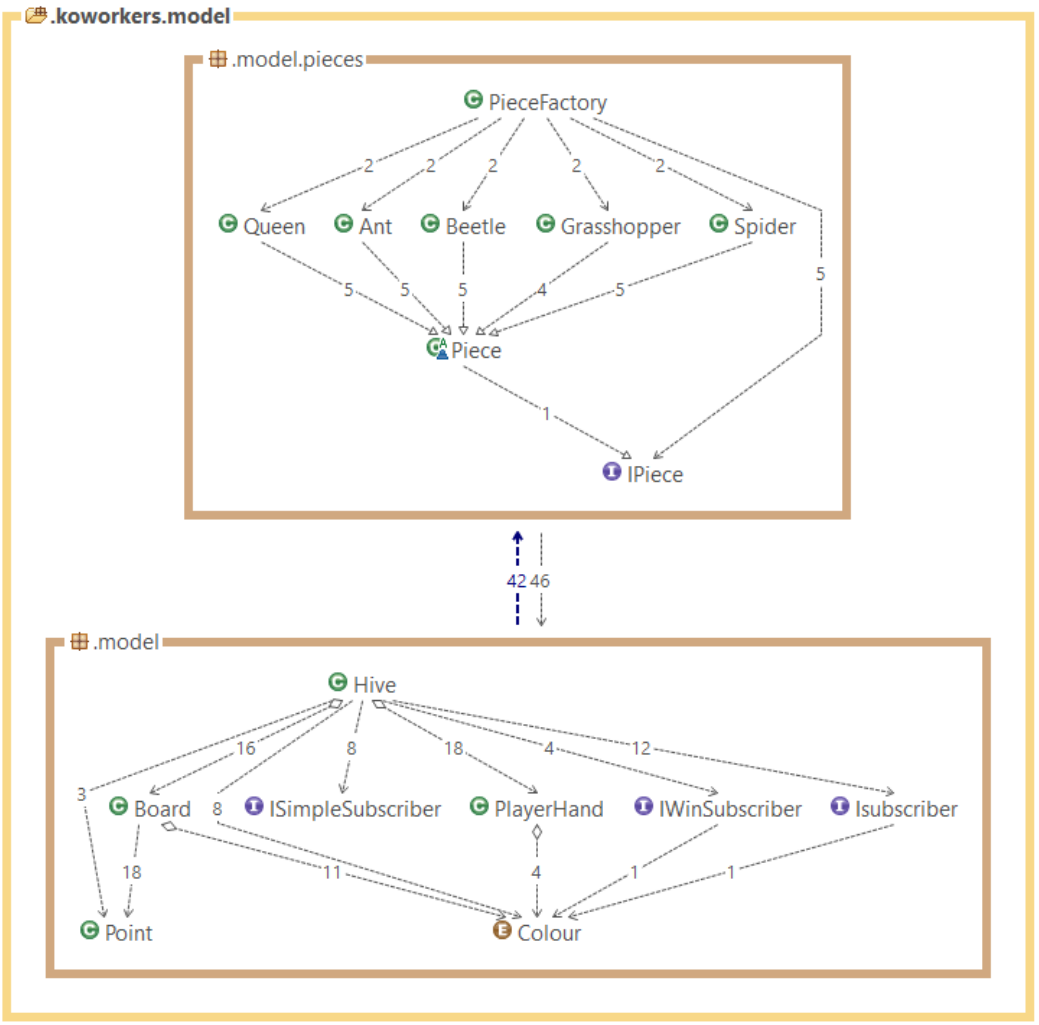
5.3 Analytiska verktyg

5.3.1 PMD

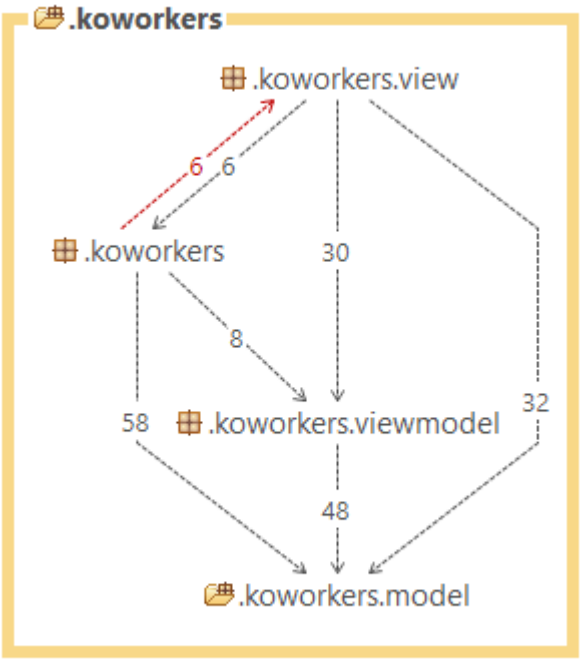
PMD är program som analyserar kod och hittar vanliga fel. Här används en plugin i Android Studio, PMDPlugin, för att analysera koden. När PMD med standardinställningar körs på projektet fås 633 problem, men majoriteten av dessa är antingen inte relevanta eller obetydliga problem. Exempel på problem som vi bortser från är för korta klassnamn, problem om hur koden dokumenterats och liknande. Ett exempel på problem som PMD tar upp som är relevanta är på kognitiv komplexitet, det vill säga hur komplicerad koden är att förstå. Det finns metoder i Hive, Beetle, Piece och PlayerHandFragment som är stora, vilket enligt PMD gör dem för komplexa. För att undvika detta hade man kunnat dela upp metoderna i mindre, vilket hade gjort metoderna mer lättförståeliga. Ett annat exempel på ett problem som PMD tar upp är att det finns många ställen där koden bryter mot Law of Demeter.

5.3.2 STAN

STAN är ett program som analyserar kod och visar strukturen, det vill säga de beroenden som finns mellan klasser och paket. I figur 4 ses strukturen på modell-paketet, och även strukturen på pieces-paketet, som är en del av modellen. I figur 5 ses strukturen på en hög nivå, det vill säga beroenden mellan paketen ‘model’, ‘view’ och ‘viewmodel’.



Figur 4: Beroenden i modellen, resultat från STAN



Figur 5: Beroenden mellan paketen model, viewmodel och view, resultat från STAN

5.4 Access control and security

N/A

6 References

6.1 Verktyg

- **Android Studio** IDE:n vi gjorde projektet med.
- **Draw.io** Ett verkyg som användes för att måla UML-diagram.
- **Figma** Ett verktyg som användes för att ta fram den grafiska designen.
- **GitHub** Lagring och versionshantering för Git av projektet.
- **Google Drive** Samlad lagring av samtliga dokument rörande projektet.
- **Slack** Kommunikationstjänst som användes för kommunikation om projektet
- **Trello** Organiseringsverktyg för de User Stories vi jobbat utifrån i projektet
- **Zoom** Kommunikationsverktyg för videosamtal för samtal om projektet.

6.2 Bibliotek

- **JUnit** används för att testa programmet.

Peer Review: Fia med knuff

av Koworkers

Koden är lätt att förstå och sätta sig in i, eftersom namn på metoder och klasser är bra, man får förståelse på vad som görs i metoderna och det är tydligt vad klasserna representerar. Koden är även väldokumenterad, några metoder saknar fortfarande dokumentering men det ser ut som att det bara inte hunnits med än. En bra sak är att även privata variabler har kommentarer som förklarar vad de representerar, vilket även det gör koden lättare att förstå. En annan sak som är väldigt bra är att koden är vältestad.

Det används inte några interfaces eller abstrakta klasser, men det finns inte riktigt ett behov för det i nuläget. På så sätt används rätt abstraktioner, då det inte är bra att till exempel ha ett interface bara för att ha det.

Designmönster

Strukturen på koden följer Model View Viewmodel, MVVM, det vill säga att vyn ska bero av viewmodel, och viewmodel ska bero av modellen, men inte i omvänd ordning. Dock implementeras den inte riktigt rätt då det ligger mycket logik i vyklasserna som egentligen borde ligga i modellen. Till exempel borde makeTurn i standardboardFragment ligga i modellen. Vyn borde egentligen bara kalla på metoder i viewmodellen hanterar omvandlingen från modellen till vyn. Sen borde viewmodellen kalla på metoder i modellen som har hand om logiken i programmet. Modellen borde sedan notifiera viewmodellen att den uppdaterats och sedan notifierar viewmodellen vyn via till exempel observer-pattern eller en eventbus. Mycket av logiken som ligger i viewmodellen skulle också kunna ligga i modellen, allt med selected piece, currentplayer, playerNames, diceValue, movablePieces etc. skulle kunna läggas i modellen istället. Vissa av dessa verkar inte ens användas i nuläget.

I koden finns en klass GameFactory som för tankarna till factory-pattern, men som inte använder ett interface. För att GameFactory skulle fylla någon funktion borde den använda sig av ett interface som Game implementerar. Ett Game-interface skulle kunna göra att man senare kan utvidga koden med olika sorters spel, men vi ser inget behov av detta och tycker att det vore rimligt att ta bort GameFactory och istället lägga logiken i konstruktorn till Game.

Designprinciper

Principen Open-Closed Principle innebär att moduler ska vara öppna för utvidgning, men stängda för ändring. Koden har många privata variabler, vilket innebär att andra klasser inte kan ändra på dem, vilket gör de stängda för ändring. Det finns flera variabler som är final, vilket är ännu bättre ur ett OCP-perspektiv, men det finns fler variabler som skulle kunna vara final, så det finns utrymme för förbättring. För att följa OCP behöver koden också vara öppen

för utvidgning vilket den inte alltid är i nuläget. Det märks främst i StandardBoardFragment där det finns flera hårdkodade listor för t.ex positioner. Vill man göra brädet större blir därför detta svårt, då fler variabler behöver hårdkodas, istället för att t.ex bara ändra en längd på en lista.

Det finns även flera getters för listor m.m som gör att man kan komma åt och ändra i listorna. Detta skulle man kunna fixa genom att, till exempel, tillämpa defensive copying. Det finns även flera metoder som inte har några access modifiers, till exempel getMovablePieces, isMovable och removePiece i Player-klassen.

Att det ligger så mycket logik i vyn och viewmodellen bryter även mot single responsibility principle vilket gör koden svårare att förstå och som det är nu så fungerar inte modellen utan vyn/viewmodellen vilket gör det svårt att återanvända modellen/koden i andra sammanhang.

Förbättringar

Funktionen connectBoardsPositionsIds skulle kunna göras om till en lista och loopas igenom istället, liknande kan också göras i connectHomePositionsIds. Alla imageViews för de möjliga positionerna borde kunna läggas i en lista vilket skulle göra det lättare att kunna återanvända koden för olika antal spelare.

Det skulle förbättra koden att ha en viewmodel för varje vy och sedan flytta logiken som för närvarande ligger i vyerna och varken hör hemma i view eller modellen där. Vyerna borde egentligen bara säga när något blivit klickat på och sedan låta viewmodellen/modellen hantera detta. Att ha en viewmodel för varje vy är också bra för att göra koden mer återanvändbar, då viewmodellen korrekt implementerad viewmodel ger uppdelad kod och det är lätt att bara ta den delen av vyn och viewmodellen man vill använda igen.

För att förbättra användarvänligheten skulle det vara bra med något som förmedlar var man kan klicka och vems tur det är.