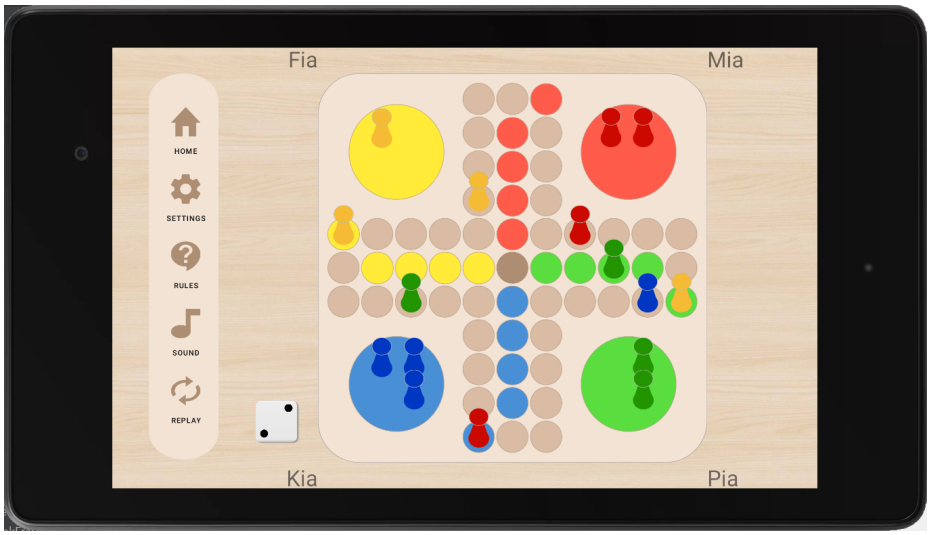
Systemdesignsdokumentation för Fia med knuff

Hanna Boquist, Amanda Cyrén, Johan Selin, Emma Stålberg, Philip Winsnes

24 oktober 2021
Version: 1.1



Innehåll

1 Introduktion

Syftet med det här dokumentet är att beskriva applikationens design och övergripande struktur som lett fram till den färdiga applikationen.

1.1 Designmål

Målet med designen är att modellen ska vara löst kopplad till vyn och vy-modellen så att det grafiska användargränssnittet kan bytas ut till en annan kontext. Vidare ska designen säkerställa att modellen inte har något beroende till externa bibliotek. Designen ska möjliggöra utökning med ytterligare funktionalitet, utan att kräva större modifiering i den redan existerande programkoden. Systemet ska vara vältestat samt kvalitétsgranskat för att säkerställa välfungerande programkod.

1.2 Definitioner, akronymer och förkortningar

- GUI, grafiskt användargränssnitt.
- MVVM, ett designmönster som används vid utveckling av applikationer innefattande ett GUI. Implicerar en struktur som består av en modell, vy och vy-modell.
- Vy, en visuell representation av gränssnittet.
- Java, ett programmeringsspråk som är plattformsoberoende.
- JUnit, ett ramverk för testning i programmeringsspråket Java.
- AssertJ, ett komplement till JUnit som förbättrar läsbarheten av testerna.
- Aktivitet, en skärm där användargränssnittet ritas upp och som användaren interagerar med.
- Fragment, tillhör en aktivitet som en modulär del av gränssnittet.
- Livscykel, syftar på tiden ett element är aktivt.
- NavHost, en behållare för navigation via en NavController.
- NavController, ett objekt som hanterar app-navigation inom en NavHost.
- CPU, en spelare som är kontrollerad av datorn och utför beräknade drag.
- Layoutfil, en fil som beskriver vad en vy ska innehålla.
- Android, ett operativsystem. Syftar även på Androids ramverk för att utveckla applikationer.

2 Systemarkitektur

Systemarkitekturen följer en MVVM-struktur vilket avskiljer applikationens logik och grundsystem från programmets GUI. Det förhindrar att modellen påverkas av applikationens livscyklar. Således blir applikationens delar väldefinierade och mer testbara inom de olika ansvarsområdena. För utveckling i Android är detta mönster rekommenderat eftersom användarna inte riskerar dataförluster om till exempel operativsystemet skulle avsluta applikationen av minnesskäl. Är applikationen bunden till nätverkstjänster fortsätter den att köra trots svag eller ingen nätverksanslutning.

För att avskilja modellen från vyn används en vy-modell, GameViewModel. Den ansvarar för att förse vyer med aktiv data i realtid från modellen. Ansvarsområden för vyerna är att lyssna på respons från användarna och presentera data i GUI:t. Vy-modellen har inga beroenden av några vyer. Om den hade det skulle det innebära att den påverkas av förändringar i vyerna eller av deras livscykler. Alla vyer får sitt ursprungliga utseende genom deras associerade layoutfil. Dessa filer innehåller information om alla vyers komponenter och attribut.

2.1 Applikationens flöde

När applikationen startas körs huvudaktiviteten i programmet, specificerad i AndroidManifest-filen, typiskt kallad för MainActivity. Detta är huvudvyn för applikationen och bär en NavHost för navigation mellan de olika fragmenten. NavHostens beteende definieras av en navigations-instruktions-fil nav_graph.xml. I huvudaktivitetens layoutfil, activity_main.xml, finns en behållare för olika fragment. NavHosten, genom sin navigations-instruktion, ger fragment-behållaren MainMenuFragment som start-fragment. Via en NavController kan applikationen navigeras mellan olika fragment.

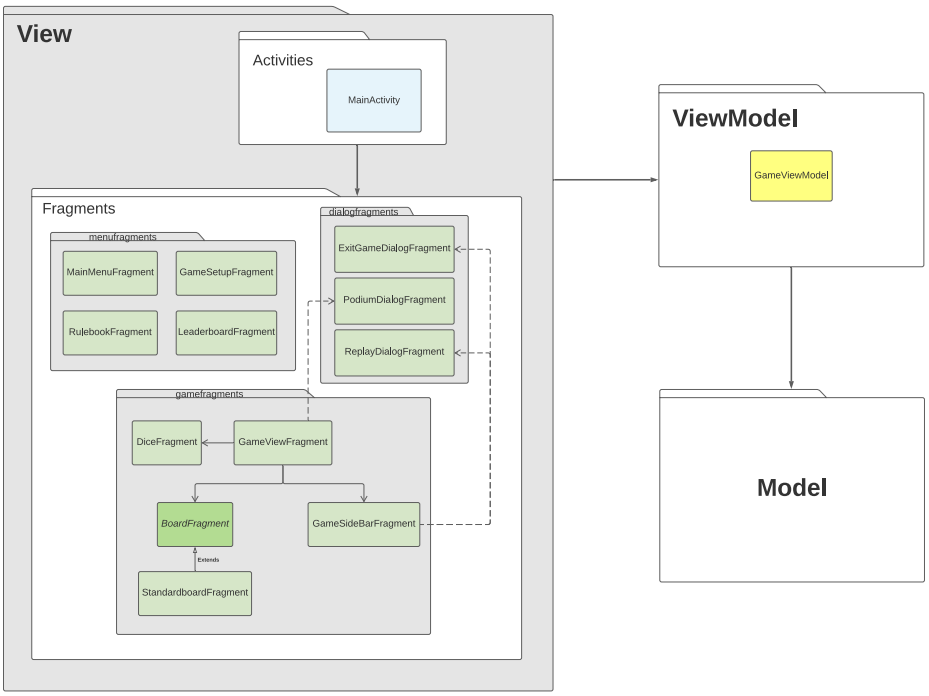
Typiskt sett så fortsätter användaren till GameSetupFragment där ett nytt spel kan startas. När inställningarna för spelet är klart så skapas och initialiseras modellen med den givna informationen och därefter navigerar programmet till GameViewFragment. I sin tur initialiserar den de nödvändiga fragmenten för ett spel: GameSideBarFragment, BoardFragment, DiceFragment. Väl i spelet så sker informationsutbytet mellan vymodellen och fragmenten via MVVM-strukturen vilket förklaras i detalj i avsnitt 3.

När ett fragment eller vy pausas, stoppas, eller stängs av så kallas deras respektiva onPause-, onStop-, onDestroy-metoder. Via dessa kan eventuell data sparas, exempelvis genom att skrivas till en databas. I nuläget sparas ingen data.

3 Systemdesign

3.1 MVVM

Model-View-ViewModel (MVVM) är ett designmönster där ett system delas upp i olika delar bestående av en modell, vyer samt vy-modeller. Modellen ansvarar för all data och information i systemet. Vyerna är de delar av systemet som presenterar datan för användaren och låter användaren interagera med programmet. Vymodellen fungerar som en förbindelse mellan modellen och vyerna och arbetar tillsammans med modellen för att få fram och spara datan.



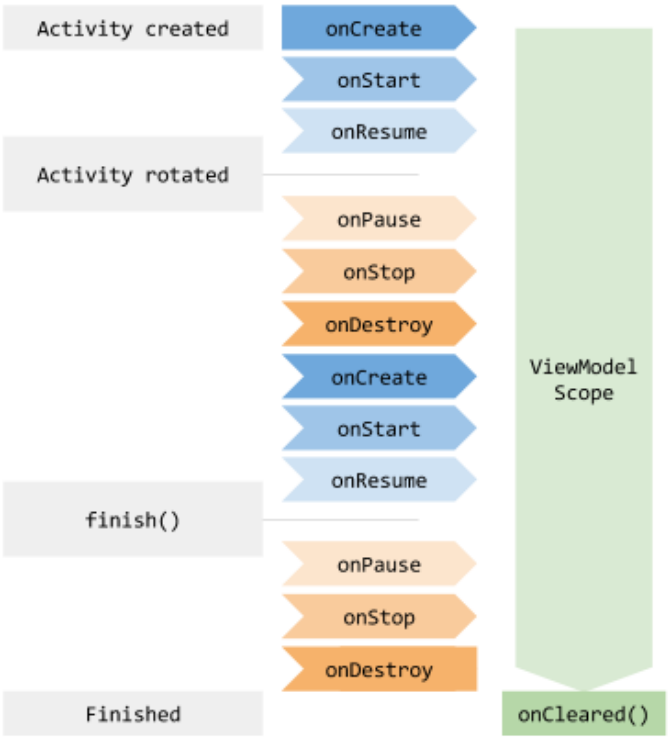
Figur 1: Designdiagram

3.1.1 Varför MVVM istället för MVC

Den allra största fördelen med MVVM-mönstret vad gäller Android-applikationer har att göra med hur livscykler hanteras i Android. Eftersom Android-ramverket är i full kontroll över UI-kontrollerna, det vill säga aktiviteter och fragment, så kan de förstöras eller återskapas utanför utvecklarens kontroll. När detta sker så förloras datan som är sparad i dem. Eftersom dessa kontroller förändras och återskapas väldigt ofta blir det en stor mängd arbete att hantera på egen hand.

Som lösning på detta problem tillhandahåller Android-ramverket en hjälpklass ”ViewModel” för att spara relevant data som en UI-kontroller behöver på ett livscykelsmedvetet vis. Klasser som ärver av ViewModel överlever livscykelförändringar och förstörs inte förrän aktiviteten eller fragmentet den tillhör slutar existera.

När en vymodell väl dör så friar Android-ramverket automatiskt upp dess resurser. En vymodells livscykel illustreras i figur 2.



Figur 2: Livscykeln hos en vymodell jämtemot dess tillhörande aktivitet. Från [1]

3.1.2 Paketrelationer

Enligt MVVM-strukturen så ska varje vy (fragment i detta fall) ha en eller flera vymodeller för att hantera dess data. Detta förbises eftersom alla vyer förutom de relaterade till spelet är så lättviktiga att de inte anses behöva en dedikerad vymodell. Datan för dessa vyer sparas temporärt som attributer i deras respektive vy-element. Detta sker automatiskt genom Android ramverket. Den enstaka vymodellen kommunicerar direkt med modellen endast genom Game-klassen där Game fungerar både som en logisk enhet och en fasad. Detta betyder att vymodellen endast har en direkt koppling till en klass i modellen och blir oberoende av den interna strukturen av modellen.

3.2 Modell

Modellen är konstruerad på så vis att den ska kunna förflyttas till valfri plattform. Genom Game-klassens offentliga metoder ges tillgång till nödvändiga funktioner för att spela Fia med knuff och eventuell information GUI:t vill ha för att rita upp en representation av spelet. Dessvärre ges direkta referenser till känsliga objekt ut vilket försvagar den strukturella säkerheten i modellen. Modellen drivs med hjälp

av MVVM-strukturen vilket innebär att endast vymodellen kommunicerar med modellen och vyn får sin information om modellen genom vymodellen.

3.2.1 Intern arkitektur

De centrala klasserna i modellen är Game och Board. Game binder ihop modellens olika komponenter till en samlad enhet som kan representera brädspelet. Board hanterar spellogiken vad gäller förflyttning av pjäser samt representerar spelplanen. Övriga klasser hjälper huvudklasserna att förminska sina ansvarsområden för att följa Separation of Concern-principen.

3.2.2 Datorspelare

Som en del av applikationen är det möjligt att välja att spela mot en eller flera CPU:er. Klassen CPU är en subklass till klassen Player och ärver därmed alla attribut som en Player har. I tillägg till det har CPU-klassen egen logik som används för att bestämma vilket drag som ska göras när det är en CPU-spelares tur.

Logiken utgår från en fördefinierad rangordning av kriterier där det är högst prioriterat att kunna slå ut en annan pjäs. Allra högst om pjäsen som slår ut dessutom är i sin hem-position, annars om de både redan är ute på brädet. Därefter prioriteras om CPU-spelaren kan gå ut med en pjäs från hemmet, alltså vid tärningsslag av en etta eller sexa. Kan inget av dessa drag göras så flyttas den pjäsen som kommit längst på brädet men ännu inte kommit in på sin individuella mittväg. I fallet då alla pjäser som kan flyttas är på mittvägen flyttas den första av spelarens pjäser.

3.2.3 Undantag

I programmet finns det två klasser som förlänger Exception-klassen:

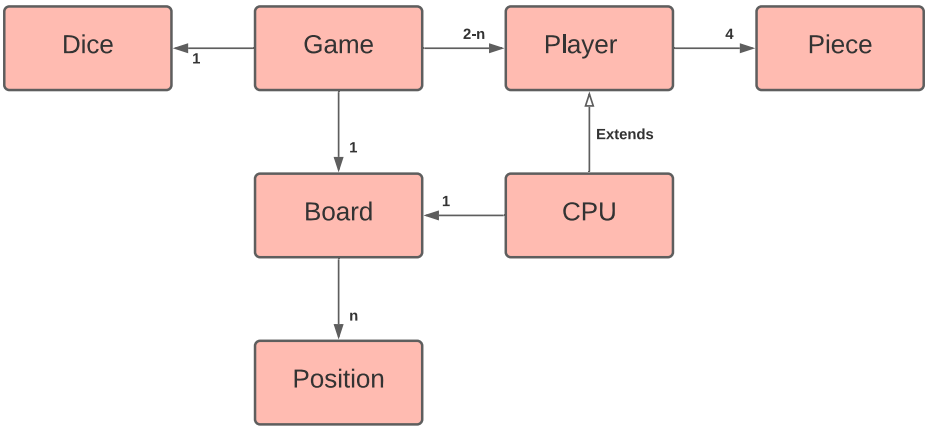
- NotImplementedException - När klienten försöker använda en funktion som ska finnas men som inte ännu är implementerad så kastas detta undantag.
- NotFoundException - Kastas när ett värde eller objekt sökes men inte kunde hittas.

I nuvarande tillstånd fångas de icke-katastrofala undantag som modellen själv kastar. Dessa blir endast loggade för tillfället men bör i en senare version även förmedlas till användaren genom GUI:t.

3.2.4 Relation mellan domänmodell och designmodell

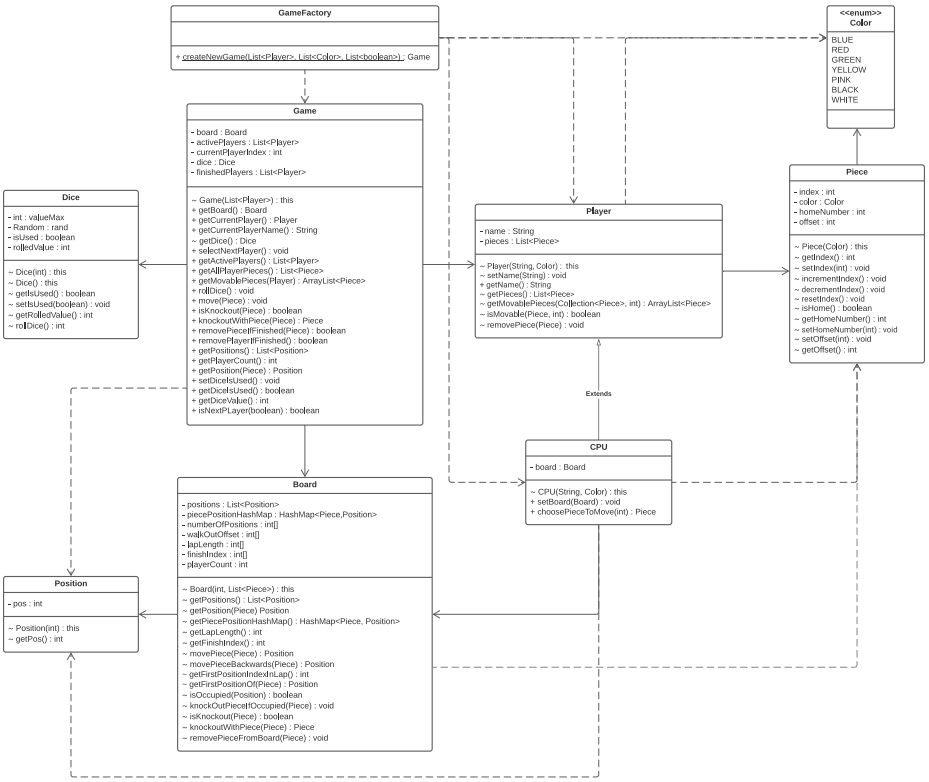
För varje klass i domänmodellen finns en motsvarande klass i designmodellen, samt några ytterligare hjälpklasser. På så vis är designmodellen starkt relaterad till domänmodellen. Designmodellen har däremot starkare kopplingar mellan vissa klasser än vad domänmodellen förmedlar, i synnerhet relationen mellan Board och Piece. Piece har ansvaret att hålla koll på hur långt den gått, var den ska gå ut, och

var dess hem är. Denna information använder Board för att beräkna var varje pjäs ska flyttas.



Figur 3: Domänmodell

Klassen GameFactory är ett ytterligare tillägg i designmodellen vars syfte är att abstrahera skapelse från användning. Den lyfter ansvaret från Game att skapa Player-objekt samt har den enda offentliga konstruktorn i modellen. Detta för att förenkla skapelsen av modellen och omöjliggöra att den får ett ogiltigt tillstånd.



Figur 4: Modelldiagram

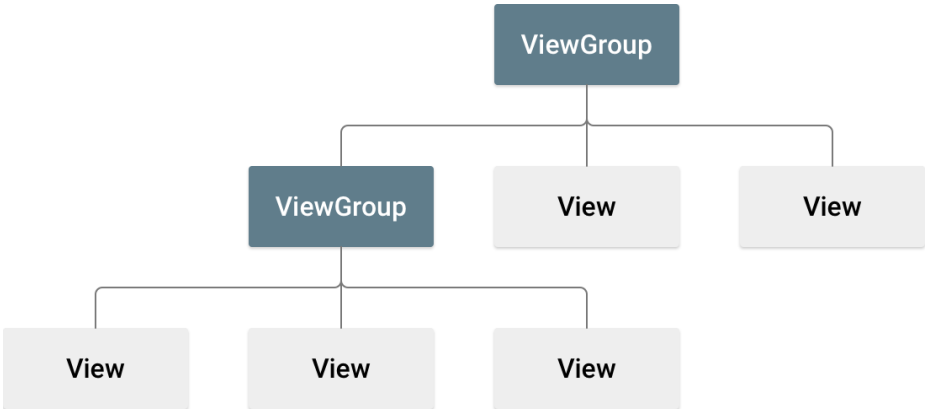
3.3 Vyn

I Android definieras gränssnittet antingen av aktiviteter och fragment. I denna design så definieras alla vyer enbart av fragment men med en singulär aktivitet som värd. Alla element i en vy definieras av layout-filer i XML-format. Vyn kommunicerar endast med vymodellen och har ingen egen logik angående spelets tillstånd. Däremot finns fortfarande gränssnitts-baserad logik i vyn. Vyn uppdateras genom att observera LiveData från vymodellen. Vad LiveData är och hur det används utvecklas vidare i avsnittet om vy-modell.

3.3.1 Layouts

Som tidigare nämnt formges gränssnittet av layout-filer. Varje aktivitet eller fragment har en egen layout-fil som i sin tur innehåller Views och ViewGroups. Gränssnittet kan även skapas och förändras programmatiskt.

- View - En View ritar typiskt upp något på skärmen som användaren kan interagera med. Dessa objekt kallar Android för ”widgets” och kommer i många former. Exempel på Views kan vara en knapp, en bild eller ett textfält.
- ViewGroup - Kallas ofta för ”layouts” och definierar strukturen för hur de View-objekt som den innefattar ska förhålla sig till varandra. En ViewGroup är också en View vilket innebär att en ViewGroup kan innehålla andra ViewGroups. Den allra populäraste och mest flexibla ViewGroup:en kallas för ConstraintLayout och används för de flesta elementen i detta projekt.



Figur 5: Exempel på en View-hierarki. Från [2]

Layoutfiler gör att logik och vy kan separeras. I layouterna specificeras hur vyn ska se ut och beteendet kontrolleras sedan av fragment-klasserna. Filerna underlättar också hanteringen av att applikationen ska anpassa sig efter enhetens storlek.

3.3.2 Aktiviteter och fragment

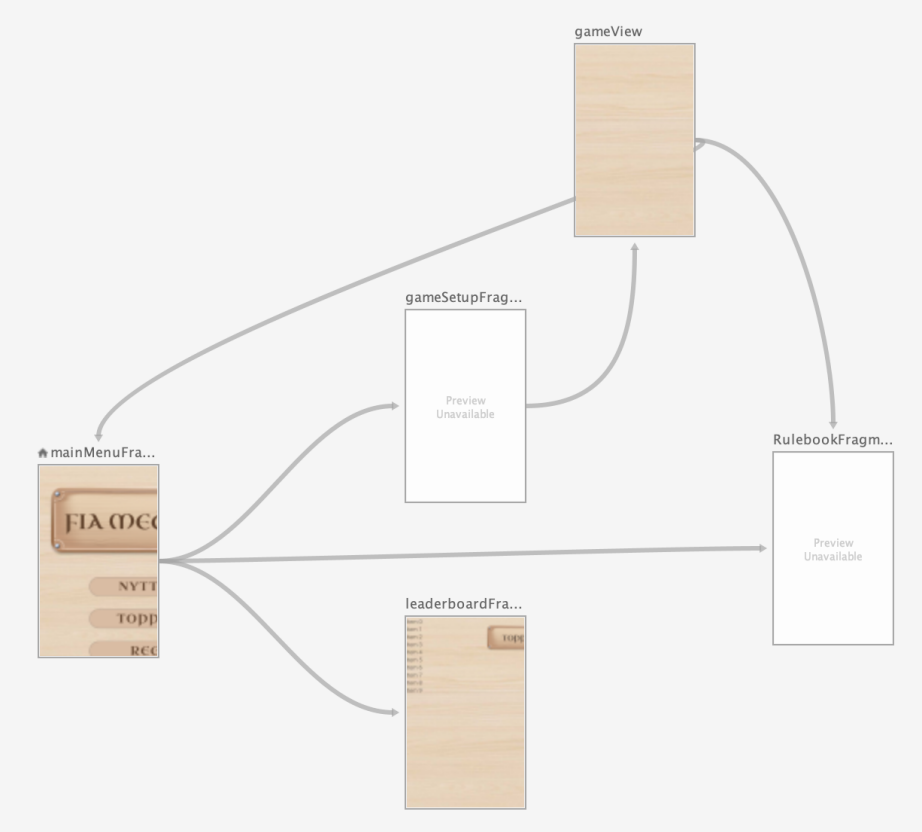
En aktivitet i Android representerar en skärm i användargränssnittet medan ett fragment även kan representera en modulär del av skärmen som kan bytas ut och ersättas med andra fragment. Flexibiliteten med fragment gör att användandet av dessa åstadkommer ett mer modulärt och återanvändbart användargränssnitt. Enligt Android är aktiviteter att föredra för element som alltid ska visas i applikationens GUI, medan fragment lämpar sig åt specifika delar av skärmen eller komponenter med helskärmsläge som ska kunna bytas ut [3].

Applikationens GUI är uppbyggt av en singulär huvud-aktivitet, MainActivity, samt ett flertal fragment för att representera de olika vyerna av programmet, exempelvis MainMenuFragment, DiceFragment och GameSideBarFragment.

BoardFragment är en abstrakt klass som innehåller alla de generella attribut och metoder som vyn för ett spelbräde inrymmer. StandarboardFragment är en subklass till BoardFragment och implementerar även de specifika egenskaper som ett standardiserat bräde besitter. Med abstraktionen av vyn för ett spelbräde kan programkoden enkelt kompletteras med ytterligare spelanpassade bräden, såsom för fler spelare. Att vyn för spelbrädet är ett fragment möjliggör ett enkelt byte från ett standardbräde till ett bräde anpassat efter specifika spelinställningar.

3.3.3 Varför endast en aktivitet istället för flera

Anledningen till en arkitektur med en singulär aktivitet och flera fragment har med det aktuella navigationssystemet att göra. En aktivitet representerar roten för en navigationsgraf, även känd som navigeringsvärd. Grafen innehåller somliga destinationer i form av fragment, se figur 6. Flera aktiviteter bidrar till flera navigationsgrafer vilket ger ett mer komplicerat navigationssystem vilket inte är nödvändigt i det aktuella systemet.



Figur 6: Navigationsgraf för destinationer i Fia med knuff

3.4 Vy-modell

Vy-modellen är lagret mellan vyn och modellen där vy-modellen representerar tillståndet av datan i modellen. Som tidigare nämnt så tillhandahåller vy-modellen LiveData i form av metoder och instansvariabler som vyn kan koppla sig till. Vy-modellen ansvarar för att konvertera och exponera UI-relaterad data i modellen till en form som är lätt för vyn att förbruka.

3.4.1 Uppdatera vy-modellerna

Vy-modellen, och i sin tur modellen, uppdateras endast på förfrågan. Dessa förfrågningar sker i form av input från användare. När en användare trycker på något modell-relaterat kallar vyn på en lämplig metod hos vy-modellen som i sin tur förmedlar förfrågan till modellen. Om en händelse inträffat som påverkar gränssnittet, uppdaterar vy-modellen motsvarande LiveData.

3.4.2 LiveData och vyerna

I dagsläget finns endast en vy-modell, GameViewModel. Den innehåller ett flertal MutableLiveData-variabler. LiveData följer observer-mönstret och är en observer-

bar klass som håller data av en specifik typ. MutableLiveData är en subklass till LiveData. Applikationens MutableLiveData-variabler sätts i vy-modellen med hjälp av metoden setValue. Dessa variabler kan observeras av vyn. De som observerar blir notifierade när variablerna ändras eller sätts, och vyn blir då uppdaterad. LiveData är även medveten om livscyklerna, vilket gör att endast aktiva vyer påverkas. Detta gör att LiveData kan uppdateras utan oro för de observerande komponenternas livscyklar. När en komponent förstörs så slutar den observera variabeln. Allokerat minne som inte används släpps då och detta förhindrar minnesläckor. Att använda sig av MutableLiveData bidrar till att MVVM följs.

Ett exempel på MutableLiveData är variabeln currentPlayer i vy-modellen. Den sätts när nästa spelare väljs. Variabeln observeras på två ställen - i BoardFragment och i GameViewFragment. Att den observeras i BoardFragment gör att applikationen kan hantera att den nuvarande spelaren är en CPU, och i så fall behövs specifika metoder köras. I GameViewFragment observeras den också, så att tärningen kan flyttas till den nuvarande spelaren.

3.4.3 Livscykel och vyerna

En vy kan enkelt och oväntat stängas ned och då försvinner dess data. Detta kan skapa problem då användare t.ex. bläddrar mellan vyer. Ett exempel från denna applikation är när en användare spelar spelet men går in i regelboken och sedan tillbaka. Då har spelvyn stängts ned och sedan skapats om, vilket gjort att datan förstörts. Som nämnts i avsnitt 3.1.1 är detta något som löses av vy-modellen, vilken gör det möjligt för vyerna att hämta tillbaka datan från vy-modellen varje gång de startas. Vy-modellen gör således att datan överlever och det blir inte längre ett problem att vyer plötsligt kan stängas ner.

3.5 Designmönster och principer

Vid objektorienterad programutveckling finns det specifika principer samt designmönster som kan tillämpas för en välstrukturerad programkod. Designmönster ger användbara lösningar på återkommande problem. Generellt så ligger SOLID-designprinciperna till grund för hela applikationen. Nedan specificeras några av de designmönstren och principer som använts.

3.5.1 MVVM

Hela applikationen bygger på designmmönstret MVVM, vilket har beskrivits utförligt ovan.

3.5.2 Observer

Designmönstret Observer innebär att ett objekt ska meddela andra objekt om event, utan att direkt bero på dem. Det observerbara objektet skickar ut händelser till en öppen kanal där intresserade kan registrera sig för att få reda på när det sker

ändringar. Som tidigare nämnts appliceras detta i vår applikation genom att vy-modellen har instanser av klassen LiveData som fungerar som den observerbara klassen. LiveData skickar ut information om vad som händer och de olika vyerna som observerar denna LiveData blir meddelade när det sker någon ändring och kan därmed uppdatera vyn till det nya tillståndet.

3.5.3 Module

Designmönstret Module handlar om att gruppera relaterade element som klasser, interfaces och metoder som en enhet. Detta följs i applikationen då klasserna är uppdelade i moduler som hör ihop. Detta medför att mindre kod behöver exponeras till hela kodbasen. Inom modellen som ligger i ett paket kan därmed många metoder vara ”package-private” vilket gör att enbart andra klasser inom modulen når dem. Även om klasser utanför inte har för avsikt att kalla på metoderna så begränsar inkapslingen onödig och riskabel exponering, vilket är en av grundprinciperna inom objektorienterad programmering.

3.5.4 Separation of Concern

Paketstrukturen är väldefinierad och separerad på liknande sätt som arkitekturen är designad. Modellen har inga beroenden av vyn och vyn behöver inte ha någon koppling till modellen för att få den uppdaterad. Modellens logik är sluten för sitt område och bidrar till en stark sammankoppling med låg koppling till vy-modellen.

Implementationsklasser hjälper med att logik inte exponeras och att vi får låga kopplingar mellan områden som modell och vy-modell. Exempel på detta är klassen GameFactory som hjälper vy-modellen att initiera stora delar av modellen. Om inte klassen hade implementerats skulle vy-modellen eller Game tvingas ha kopplingar till Player och Color, samt hålla GameFactorys logik i en egen metod. Detta bidrar till mer ansvarsområden än vad som är nödvändigt. GameFactory avlastar alltså instansieringen av modellen samtidigt som mindre kopplingar binds mellan vy-modellen och modellen.

3.5.5 Liskov Substitution Principle

Arkitekturen är förberedd för att hantera och skapa olika subklasser av bräden för att kunna hantera dem utan hänsyn på vilket specifikt bräde de är. Det bidrar till att hanteringen av bräden blir modulärt och samma förväntningar kan antas för alla framtida bräden. BoardFragment är en abstrakt klass som delar gemensamma egenskaper för alla klasser som ärver BoardFragment. Samma egenskaper finns för Player och CPU. Överallt i applikationen kan en CPU behandlas som en Player. CPU ärver egenskaper av Player utan att påverka dem. CPU utökar endast funktionalitet vilket gör att principen uppfylls.

3.6 Hantering av beständig data

Bilder och vektorer på exempelvis tärningen, spelpjäserna och olika knappar laddas in då applikationen startas och lagras i form av png- och xml-filer.

Det finns i nuläget inget behov i applikationen att kunna spara användare. Vid utökningar av applikationen, exempelvis för att kunna se statistik samt en sparad topplista så skulle data om användarna behöva sparas mellan olika spelomgångar.

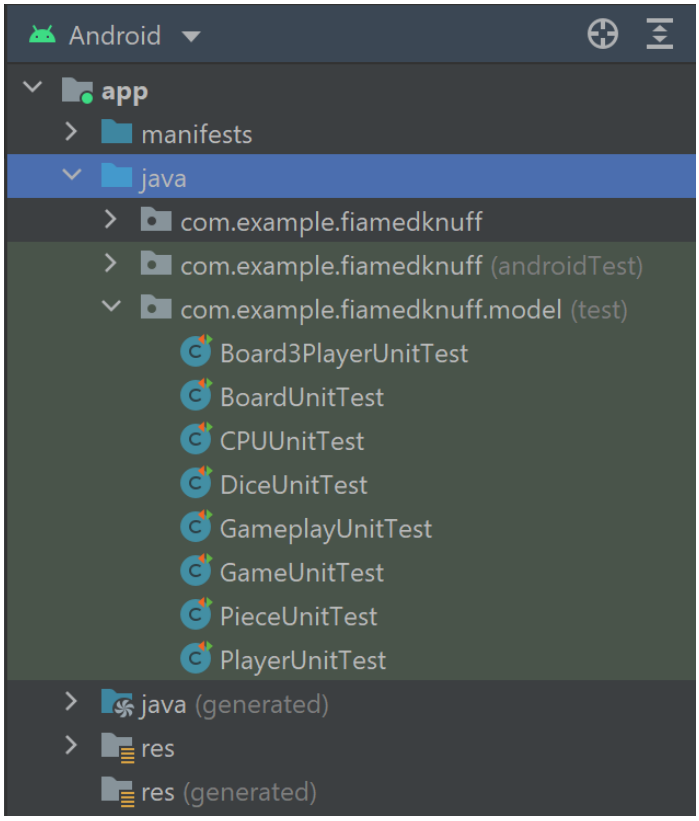
För utökningen att kunna stänga ner ett pågående spel och ta upp det senare behöver det aktuella tillståndet av spelet att sparas, så att spelet vid återstart fortsätter från samma läge som när det pausades eller stängdes ner.

4 Kvalité

Vid mjukvaruutveckling är det essentiellt att tester och kvalitetssäkring genomförs för att säkerställa en väl fungerande programkod med avsaknad av buggar, som uppfyller kundens krav.

4.1 Testning

Modellen i applikationen har fullt ut testats med JUnit. Alla offentliga metoder testas via JUnit och testar implicit klassernas privata metoder. Övergripande så testas var klass för sig och kontrollerar att all alla metoder fungerar som förväntat och att de initialiseras korrekt. Två mindre klasser, Color och Position, testas inte för sig, men integreras i resterande tester. Ytterligare testas hela spelmodellen som en enhet där vi simulerar en spelomgång via kod. Testerna hittas här alternativt i Android Studio enligt figur 7.



Figur 7: Testernas sökväg i Android Studio

4.1.1 Kontinuerlig integrering

Kontinuerlig integrering används genom plattformen CircleCI för att automatisera tester. Detta möjliggör snabb identifiering av problem som kan uppstå vid sam-

manställning av samtliga gruppmedlemmars arbete. Projektet på CircleCI hittas här.

4.2 Kända problem

- Applikationens grafiska användargränssnitt stöds endast av ett begränsat antal skärmstorlekar. Pixeldensitet 320dp eller 480dp säkerställer den avsedda grafiska användarupplevelsen medan pixeldensitet som avviker från dessa siffror kan deformera användargränssnittet.
- När en pjäs vänder i mitten så kan den ställa sig på en ruta som är ockuperad av samma spelare och på så vis knuffa tillbaka en egen pjäs.
- När CPU ska göra sitt drag är det ingen fördröjning vilket gör att det går snabbare än GUI:t kan uppdatera.
- CPU:ns tärningsslag är opålitigt och kan påverka efterkommande spelares drag.
- Det delas ut referenser till muterbara objekt till klasser som inte ska ändra objekten.
-

4.3 Kvalitetsgranskning

Kodbasen har analyserats med verktyget PMD för att försäkra att kodstilen är konsekvent och kvalitén hålls hög.

5 Referenser

5.1 Verktyg

- Android Studio Projektets IDE.
- Figma En applikation för design av grafiska användargränssnitt.
- GitHub Versionshantering med hjälp av Git.
- Trello Verktyg för att kunna planera och visualisera status i projektet.
- Lucid Ett verktyg för att skapa och jobba med UML-diagram online.
- CircleCI Verktyg för kontinuerlig integrering.
- Google Drive En delad mapp för att ha mötesprotokoll och dokument samlade.
- Slack Plattform för kommunikation inom projektet.
- Zoom Verktyg för distansmöten.
- PMD Verktyg för statisk kodanalys

5.2 Bibliotek

- JUnit Javas ramverk för "unit-testing".
- AssertJ Ytterligare assertions för JUnit och förbättrar läsbarheten av tester.
- Apache Commons Underlätta serialisering.
- numAndroidPageCurlEffect Används för att skapa en effekt av att vända ett blad i regelboken.

Källhänvisning

[1] Android Developers. ”Layouts”, 2021. [Online]. Tillgänglig: https://developer.android.com/topic/libraries/architecture/viewmodel (hämtad: 2021-10-20).

[2] Android Developers. ”Layouts”, 2021. [Online]. Tillgänglig: https://developer.android.com/guide/topics/ui/declaring-layout (hämtad: 2021-10-20).

[3] Android Developers. ”Fragments”, 2020. [Online]. Tillgänglig: https://developer.android.com/guide/fragments (hämtad: 2021-10-20).