

Krav- och analysdokumentation för Fia med knuff

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

HEJAP

TDA367 Objektorienterat programmeringsprojekt
Chalmers tekniska högskola
24 oktober 2021

Innehåll

1	Introduktion	1
1.1	Generella egenskaper för applikationen	1
1.2	Definitioner, akronymer och förkortningar	1
2	Krav	2
2.1	Användarberättelser	2
2.2	Definition of Done	10
3	Användargränsnitt	11
3.1	Startsidan	11
3.2	Skapelsen av ett spel	12
3.3	Spelvyn	12
3.4	Regelboken	14
3.5	Topplista	14
4	Domänmodell	15
4.1	Klassansvar	15
5	Referenser	16
5.1	Verktyg	16
5.2	Bibliotek	16

1 Introduktion

Projektet ämnar att skapa en Android-applikation av brädspelet Fia med knuff, som tillåter en eller flera spelare att spela mot en CPU eller mot varandra. Applikationen ska tillhandahålla spelaren ett konventionellt Fia med knuff i digitalt format men också erbjuda extra funktionalitet såsom lokalt flerspelarläge och anpassningar av spelet efter antalet spelare.

1.1 Generella egenskaper för applikationen

- Programmet är huvudsakligen utvecklat för Android-läsplattor.
- Programmet är skrivet i Java.
- Spelet är en digital tolkning av ett konventionellt Fia med knuff. Spelet öppnar dessutom upp för eventuella speciallägen som skiljer sig från ett traditionellt Fia.
- Antalet spelare är valbart.
- Spelet hjälper visuellt till med tillåtna drag och turordning.
- En regelbok hjälper spelare att förstå hur spelet ska spelas.
- Spelare kan välja att spela mot en eller flera CPU:er.
- Under spelets gång är det valbart att lyssna på stämningsfull musik.
- Spelet har stöd för både svenska och engelska.

1.2 Definitioner, akronymer och förkortningar

- GUI, grafiskt användargränssnitt.
- Java, ett programmeringsspråk som är plattformsberoende.
- JUnit, ett ramverk för testning i programmeringsspråket Java.
- JavaDoc, ett standardiserat sätt att skapa dokumentation till javaprojekt.
- UML-diagram, olika diagram som visar en översikt på kodstrukturen.
- Aktivitet, en skärm där användargränssnittet ritas upp och som användaren interagerar med.
- Fragment, tillhör en aktivitet som en modulär del av gränssnittet.
- CPU, en spelare som är kontrollerad av datorn och utför beräknade drag.
- Vy, en visuell representation av gränssnittet.
- Android, ett mobilt operativsystem.

2 Krav

2.1 Användarberättelser

Prioritet 1	Prioritet 2	Prioritet 3	Prioritet 4
US01: Rita upp spelplan	US10: Spela mot en CPU	US16: Feedback när en pjäs går ut	US19: Spelresultat
US02: Slå tärning	US11: Avsluta ett pågående spel	US17: Spelet går att anpassa efter olika enheter	US20: Sätta på/stänga av musik
US03: Välja mellan drag	US12: Starta om ett pågående spel	US18: Feedback vid vinst	US21: Topplista
US04: Endast göra tillåtna drag	US13: Läsa regler för spelet		US22: Spela fler än 4 spelare
US05: Urskilja pjäser	US14: Pjäs hoppar stevvis framåt		US23: Timer
US06: Välja antal spelare	US15: Huvudmeny		US24: Spelläge med specialregler
US07: Se vilken spelares tur			US25: Anpassa svårighetsgrad
US08: Turen går vidare till nästa spelare			US26: Välja tema
US09: Knuffa ut pjäs			

Figur 1: Färdiga User Stories är markerade i grönt

Prioritet 1

US01: Rita upp spelplan

Implementerat?

- Ja

Beskrivning

- Som spelare vill jag kunna se spelplanen för att kunna spela spelet.

Acceptanskriterier

- Spelbrädet ska synas.
- Varje spelare ska ha varsin färg.
- Varje spelare ska ha fyra pjäser i sin färg.
- Det ska synas hur många spelare som är med.
- Man ska se pjäserna för varje spelare.

US02: Slå tärning

Implementerat?

- Ja

Beskrivning

- Som spelare vill jag kunna slå en tärning så att jag kan göra ett drag.

Acceptanskriterier

- Spelaren kan slå en tärning.
- Bara den aktiva spelaren kan använda tärningen.
- Efter slaget kan spelaren göra ett drag enligt resultatet på tärningen.
- Spelaren kan bara slå tärningen en gång.

US03: Välja mellan drag

Implementerat?

- Ja

Beskrivning

- Som spelare vill jag kunna välja mellan olika drag för att flytta mina pjäser och ta mig fram i spelet.

Acceptanskriterier

- Spelaren ska bara kunna välja att flytta sina egna pjäser.
- Spelaren ska inte kunna flytta en pjäs som inte går att flytta.
- Efter slaget kan spelaren göra ett drag enligt resultatet på tärningen.
- Spelaren kan välja vilken av sina pjäser som den vill flytta.

US04: Endast göra tillåtna drag

Implementerat?

- Ja

Beskrivning

- Som spelare vill jag endast kunna utföra tillåtna drag så att jag spelar enligt spelets regler.

Acceptanskriterier

- Ett otillåtet drag kan ej genomföras.
- En spelare som står i boet ska inte kunna gå ut på en 3:a.

US05: Urskilja pjäser

Implementerat?

- Ja

Beskrivning

- Som spelare vill jag kunna urskilja mina pjäser från de andra för att kunna förstå spelet lättare.

Acceptanskriterier

- Alla spelares pjäser har en och samma färg.
- Flera spelares pjäser kan inte ha samma färg.
- Pjäserna syns på brädet.

US06: Välja antal spelare

Implementerat?

- Ja

Beskrivning

- Som spelare vill jag kunna välja antalet spelare (2-4st) så att spelet kan anpassas utefter det.

Acceptanskriterier

- Det ska gå att välja antalet spelare.
- Det ska inte gå att välja att det är fler spelare än 4.
- spelet ska anpassas utefter antalet spelare som är valda, dvs antalet pjäser = $4^*\text{antalet spelare}$ och antalet bon är lika många som antalet spelare.
- Varje spelare har tilldelats en färg med tillhörande pjäser.

US07: Se vilken spelares tur

Implementerat?

- Ja

Beskrivning

- Som spelare vill jag kunna se vilken spelares tur det är så att jag kan följa spelet.

Acceptanskriterier

- Tärningen placeras vid aktiv spelare.
- Endast en spelare ska vara aktiv i taget.

US08: Turen går vidare till nästa spelare

Implementerat?

- Ja

Beskrivning

- Som spelare vill jag att turen att spela går över till nästa person när jag gjort mitt drag så att spelet kan fortsätta.

Acceptanskriterier

- Den aktiva spelaren ska ha en tärning vid sitt namn.
- Turen ska gå vidare till nästa spelare när en spelare gjort ett drag.
- Nästa person att spela kan aldrig bli någon som redan gått ut med alla sina pjäser.
- Om inget drag är möjligt ska turen gå över direkt till nästa spelare.
- nästa person att spela kan aldrig bli någon som redan gått ut med alla sina pjäser.
- Om inget drag är möjligt ska turen gå över direkt till nästa spelare.

US09: Knuffa ut pjäs

Implementerat?

- Ja

Beskrivning

- Som spelare vill jag kunna knuffa ut en annan spelares pjäs.

Acceptanskriterier

- Pjäsen som blir knuffad flyttas tillbaka till sitt bo.
- Man inte knuffa ut en egen pjäs.
- Om en pjäs hamnar på samma position som en annan pjäs så ska den pjäsen knuffas ut.
- Det ska synas visuellt när en pjäs knuffas ut.

Prioritet 2

US10: Spela mot en CPU

Implementerat?

- Delvis

Beskrivning

- Som spelare vill jag kunna spela mot en CPU så att jag kan spela när jag är ensam.

Acceptanskriterier

- Spelaren ska kunna välja att spela mot datorn.
- CPU-spelaren sköter sig själv under spelets gång.
- CPU-spelaren har logik som gör att den väljer smarta drag.
- CPU-spelarens drag visas upp visuellt i vyn.

US11: Avsluta ett pågående spel

Implementerat?

- Delvis

Beskrivning

- Som spelare vill jag enkelt kunna avsluta spelet så att jag inte är fast i spelet för länge.

Acceptanskriterier

- Tillbakaknapp i sidopanelen för att komma till huvudmenyn från spelvyn.
- Dialogruta som ber om att bekräfta valet.
- Det ska finnas ett val att spara det pågående spelet för att senare återuppta spelet (feature).
- Det ska gå att avsluta ett pågående spel för att sedan starta ett nytt utan oönskade konsekvenser.
- Efter ett avklarat spel ska spelarna få chansen att återvända till huvudmenyn.

US12: Starta om ett pågående spel

Implementerat?

- Delvis

Beskrivning

- Som spelare vill jag kunna starta om spelet för att smidigt fortsätta spela.

Acceptanskriterier

- Knapp i sidopanelen som startar om samma pågående spel.
- Dialogruta som bekräftar valet.
- Spelplanen återställs med samma spelare. Om en spelare gått ut med sina pjäser ska också den spelaren återställas. Tärningen hamnar i sin startposition och turen hamnar hos den första spelaren.
- Efter ett avklarat spel ska spelarna få chansen att starta om spelet.

US13: Läsa regler för spelet

Implementerat?

- Ja

Beskrivning

- Som spelare vill jag kunna läsa reglerna så att jag vet hur jag ska spela.

Acceptanskriterier

- Regelboken kan nås från menyn.
- Regelboken kan nås från spelet.
- Man kommer tillbaka till huvudmenyn / spelet när regelboken stängs.

US14: Pjäs hoppar stegvis framåt

Implementerat?

- Ja

Beskrivning

- Som spelare vill jag kunna se min pjäs hoppa dit jag vill så att det blir mer verklighets-troget.

Acceptanskriterier

- Någon form av animation för flyttning av pjäser ska vara verklighetstroget och snyggt.
- Pjäsen kommer inte direkt fram till sin slutposition.
- Pjäsen försvinner från rutan den tidigare stått på när den flyttas/dyker upp på en ny.

US15: Huvudmeny

Implementerat?

- Ja

Beskrivning

- Som spelare vill jag ha en huvudmeny med knappar för nytt spel, regler och topplista för att hålla spelet organiserat.

Acceptanskriterier

- En huvudmeny med knappar för nytt spel, regler och topplista ska finnas.

- Knapparna i huvudmenyn ska vara klickbara.

Prioritet 3

US16: Feedback när en pjäs går ut

Implementerat?

- Ja

Beskrivning

- Som spelare vill jag ha någon sorts feedback när jag går ut med en pjäs så att jag känner mig nöjd.

Acceptanskriterier

- Spelare ska kunna gå ut med en pjäs.
- Kolla så att draget är tillåtet.
- Pjäsen försvinner från spelplanen.
- Grafisk feedback visas upp.

US17: Spelet går att anpassa efter olika enheter

Implementerat?

- Ja

Beskrivning

- Som spelare vill jag att det ska finnas stöd för att kunna anpassa spelet efter flera enheter.

Acceptanskriterier

- Innehållet ska skalas om proportionerligt på två valda storlekar (320dp och 480dp).
- Alla värden ska vara satta med hjälp av dimensionsfiler, så att det är enkelt att lägga till värden för en ny enhetsstorlek.

US18: Feedback vid vinst

Implementerat?

- Delvis

Beskrivning

- Som spelare vill jag ha någon feedback när jag vinner så att jag känner mig nöjd och vill vinna igen (spela igen).

Acceptanskriterier

- Spelare ska kunna gå ut med sista pjäsen.
- Kolla så att draget är tillåtet.
- Pjäsen försvinner.
- Grafisk feedback om vinst visas.
- Dialogruta dyker upp med meddelande.
- Val i dialogrutan leder till passande utgång.

Prioritet 4

US19: Spelresultat

Implementerat?

- Delvis

Beskrivning

- Som spelare vill jag att att det ska finnas en lista som visar resultatet av spelet så att jag kan se vilken ordning vi kom på.

Acceptanskriterier

- En lista ska finnas med slutordningen på spelet.
- Listan med slutordningen ska visas i slutet av spelet.

US20: Sätta på/av musik

Implementerat?

- Ja

Beskrivning

- Som spelare vill jag kunna sätta på/stänga av spelmusik så att jag kan sätta stämningen.

Acceptanskriterier

- Knapp för volym finns tillgänglig.
- Knappen för volym är klickbar.
- Vid klick på ljudknapp ska ljudet stängas av och på.
- Symbolen för ljud visar tydligt om ljudet är på eller av.

US21: Topplista

Implementerat?

- Delvis

Beskrivning

- Som spelare vill jag kunna se en topplista med tidigare vinnare och hur många gånger de vunnit, så att jag kan komma ihåg hur det har gått i tidigare matcher.

Acceptanskriterier

- Det ska finnas en topplista.
- Topplistan ska uppdateras efter avslutat spel.
- Topplistan ska gå att nås från huvudmenyn.

US22: Spela fler än 4 spelare

Implementerat?

- Delvis

Beskrivning

- Som spelare vill vi kunna spela fler än fyra spelare så att vi kan vara fler tillsammans.

Acceptanskriterier

- Man kan välja antal spelare.
- Negativa eller för stora spelarantal ska inte vara möjligt.
- Spelplanen ska anpassas för olika antal (ex. 5 spelare ger en 5-stjärna).

US23: Timer

Implementerat?

- Nej

Beskrivning

- Som spelare vill jag kunna se en timer för att se hur lång tid spelet har pågått.

Acceptanskriterier

- Synlig timer.
- När en spelomgång startar så startar timern.
- När en spelomgång är slut så stannar timern.
- Timern pausas om man går ur spelet på något vis.
- Timern pausas om man går ur spelet på något vis.
- Timern pausas om man läser reglerna eller öppnar hjälpguiden.

US24: Spelläge med specialregler

Implementerat?

- Nej

Beskrivning

- Som spelare vill jag kunna ändra spelläget till ett spel med mer specialregler.

Acceptanskriterier

- Specialregler ska kunna väljas från startkonfigurationen av nytt spel samt under spelets gång.
- Specialreglerna ska inte vara på som standard och fördefinerade värden ska vara ifyllda vid start av ett nytt spel för att underlätta och förenkla.

US25: Anpassa svårighetsgrad

Implementerat?

- Nej

Beskrivning

- Som spelare vill jag kunna välja funktionaliteter så att jag kan anpassa svårighetsgraden och välja vilken utmaning jag vill ha.

Acceptanskriterier

- Fördefinierade inställningar och värden ska redan vara ifyllda vid skapandet av ett nytt spel för att underlätta för spelarna.
- Inställningarna ska kunna påverka svårighetsgraden för spelet.

- Det finns en meny för att välja extra inställningar.
- Det ska gå att välja att se vilka följer ett drag får (visa var pjäsen landar).

US26: Välja tema

Implementerat?

- Nej

Beskrivning

- Som spelare vill jag kunna sätta ett tema så att spelet passar min stil.

Acceptanskriterier

- Knapp för val av tema finns tillgänglig.
- Knapp för val av tema är klickbar.
- Möjliga val av tema dyker upp i t.ex. en rullgardinsmeny.
- Vid val av tema ändras speldesignen (färg/tysnitt med mera).

2.2 Definition of Done

För alla användarberättelser gäller följande acceptanskriterier:

- Applikationen bygger och kompilerar.
- All implementerad funktionalitet har klarat JUnit-testerna.
- Hela kodutvecklingen är dokumenterad via versionskontroll.
- Alla specifika acceptanskriterier för de olika användarberättelserna ska vara uppnådda.
- Alla publika metoder och klasser i kodbasen är dokumenterade via JavaDoc.
- Kodstrukturen ska överensstämma med samtliga UML-diagram.
- All kod är kontrollerad av minst två gruppmedlemmar.
- Koden är referentgranskad.

3 Användargränsnitt

Syftet med den här sektionen är att beskriva och visualisera applikationens användargränssnitt med fokus på navigationen mellan programmets olika vyer. Målet med applikationens användargränssnitt är att det ska vara användarvänligt samt föra tankarna till traditionella brädspel.

Generella användarvänliga designdrag är att knappar visar sin klickbarhet genom dimning vid klick. Texter följer i stora drag samma typsnitt vilket bidrar till en enhetlig design. Beroende på enhetens språkinställningar ändras översättning mellan svenska och engelska. Eftersom applikationen är designad för Android, tar navigationen hänsyn till de konventionella knapparna som finns på en Android-enhet. Därför visas inga ytterligare visuella tillbakaknappar.

3.1 Startsidan

Vid start av applikationen nås startsidan (se figur 2) där användaren har möjlighet att antingen starta ett nytt spel, kolla topplistan eller läsa på regler för spelet.



Figur 2: Huvudmenyn

3.2 Skapelsen av ett spel

När användaren trycker på knappen ”nytt spel” på startsidan, öppnas menyn i figur 3. Där väljs antal spelare, namn på spelarna samt om någon av spelarna ska fungera som en CPU.

För att ge spelarna en förståelse över hur spelet kommer sättas upp, ändras listan med spelare dynamiskt utifrån det angivna spelarantalet. För varje spelare finns en fördefinierad färg som visar vilken färg som kommer tilldelas spelaren. För att en spelare ska kontrolleras av programmet, bockas CPU-rutan i för den spelaren. CPU-spelaren kan då dessutom tilldelas ett önskat namn. Spelet går inte att starta om några av spelarna har samma namn.

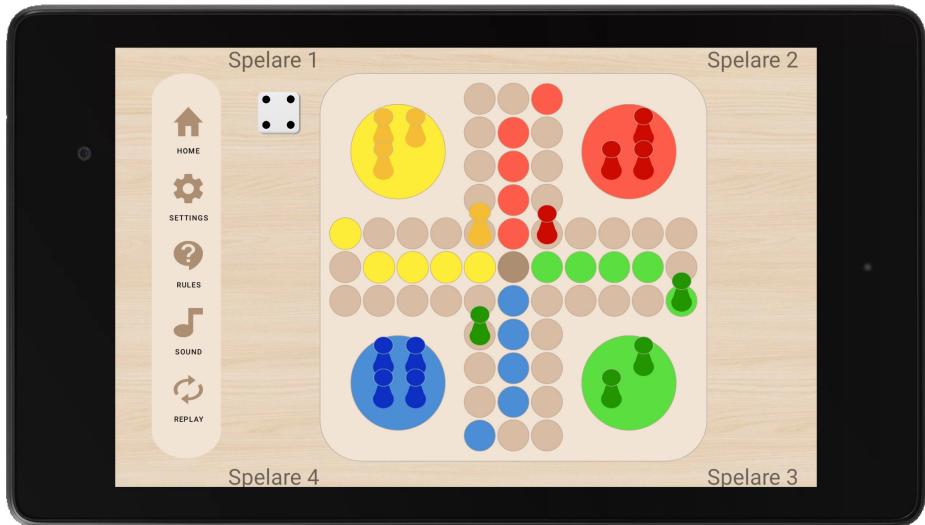


Figur 3: Skapelsen av ett spel

3.3 Spelvyn

När spelet sätts igång kommer användaren till spelvyn (se figur 4) där spelplanen ritas upp med pjäser utifrån hur många spelare som valts. Till vänster finns en sidomeny med olika navigeringsmöjligheter. Tärningen placeras vid start hos spelaren som ska börja slå och flyttas sedan runt till de olika spelarna.

Beroende på vem av spelarnas tur det är hamnar tärningen hos spelarens namn. Det signalerar för spelaren att det är spelarens tur. Då tärningen slås visas en animering vilket ger en realistisk känsla av tärningen. Därefter oscillerar de pjäser som är flyttbara för spelaren. När väl en pjäs väljs, vandrar pjäsen till den ruta den hamnar på, vilket ger ett realistiskt intryck på spelet. Dessa designval bidrar till en enklare förståelse och ett flyt i spelkänslan.



Figur 4: Spelvyn

En sidopanel låter användaren gå till hemskärmen, inställningar, regelboken, tysta bakgrundsmusiken och spela spelet från början. Vid val att gå till hemskärmen visas en dialogruta (se vänster i figur 5). Detta gäller även för valet att spela om spelet (se höger i figur 5).



Figur 5: Dialogrutor

3.4 Regelboken

Regelboken hjälper användaren med att få en förståelse över reglerna i spelet, och kan ses i figur 6. Vyn går att nå från både sidopanelen i spelvyn och i huvudmenyn. Regelboken går att bläddra i på liknande sätt man bläddrar i en fysisk bok.



Figur 6: Regelboken

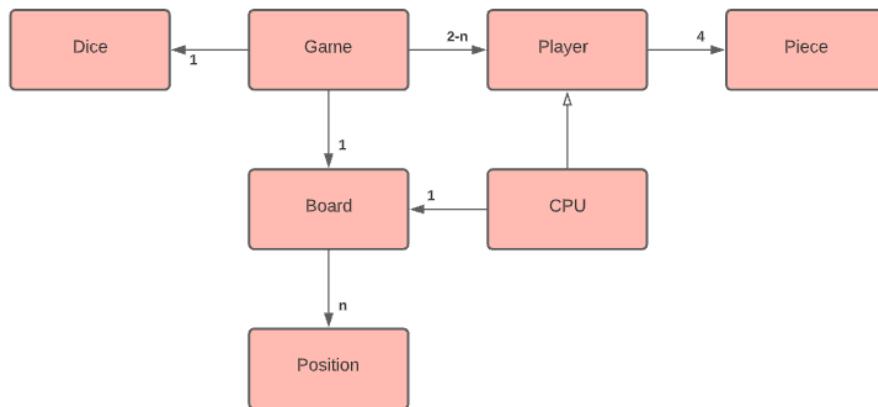
3.5 Topplista

Topplistan visar spelnamnen på de spelare som vunnit tidigare matcher. Den som har vunnit flest matcher ligger överst i listan. Topplistan ses i figur 7.



Figur 7: Topplista

4 Domänmodell



Figur 8: Domänmodell

4.1 Klassansvar

- **Dice** - Skapar en tärning som går att slå samt håller värdet av tärningsslaget.
- **Game** - Har huvudansvar över hur ett spel fungerar genom att koppla ihop de olika komponenterna som finns med.
- **Player** - Skapar och representerar spelare i modellen och ansvarar för att skapa sina pjäser.
- **Piece** - Representerar en pjäs i spelet och urskiljer sig visuellt med ett färg-attribut. Ansvarar för data om var den ska gå ut på brädet och hur många steg den tagit.
- **Board** - Skapar och representerar ett bräde av spelet. Ansvarar för att flytta alla pjäser och hålla koll på var de står.
- **CPU** - En klass som skapar CPU-objekt samt kopplar ihop spelaren med datorlogik för hur en spelare kan välja sina drag.
- **Position** - Skapar och håller ett index vilket används för att förbinda en plats i GUI:t med en position.

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 webbverktyg 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.

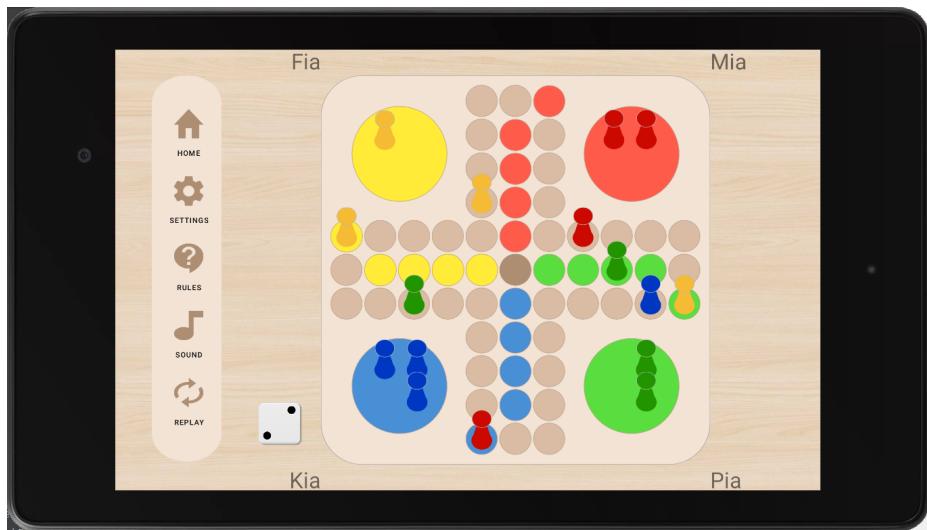
5.2 Bibliotek

- [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.

Systemdesignsdokumentation för Fia med knuff

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

24 oktober 2021
Version: 1.1



Innehåll

1	Introduktion	1
1.1	Designmål	1
1.2	Definitioner, akronymer och förkortningar	1
2	Systemarkitektur	2
2.1	Applikationens flöde	2
3	Systemdesign	3
3.1	MVVM	3
3.1.1	Varför MVVM istället för MVC	3
3.1.2	Paketrelationer	4
3.2	Modell	4
3.2.1	Intern arkitektur	5
3.2.2	Datorspelare	5
3.2.3	Undantag	5
3.2.4	Relation mellan domänmodell och designmodell	5
3.3	Vyn	7
3.3.1	Layouts	7
3.3.2	Aktiviteter och fragment	8
3.3.3	Varför endast en aktivitet istället för flera	8
3.4	Vy-modell	9
3.4.1	Uppdatera vy-modellerna	9
3.4.2	LiveData och vyerna	9
3.4.3	Livscykel och vyerna	10
3.5	Designmönster och principer	10
3.5.1	MVVM	10
3.5.2	Observer	10
3.5.3	Module	11
3.5.4	Separation of Concern	11
3.5.5	Liskov Substitution Principle	11
3.6	Hantering av beständig data	12
4	Kvalité	13
4.1	Testning	13
4.1.1	Kontinuerlig integrering	13
4.2	Kända problem	14
4.3	Kvalitetsgranskning	14
5	Referenser	15
5.1	Verktyg	15
5.2	Bibliotek	15

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 välfestat 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 innehållande ett GUI. Impliciterar en struktur som består av en modell, vy och vy-modell.
- Vy, en visuell representation av gränssnittet.
- Java, ett programmeringsspråk som är plattformsberoende.
- 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 grundsyste m från program mets GUI. Det förhindrar att modellen påverkas av applikationens livscykler. 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.

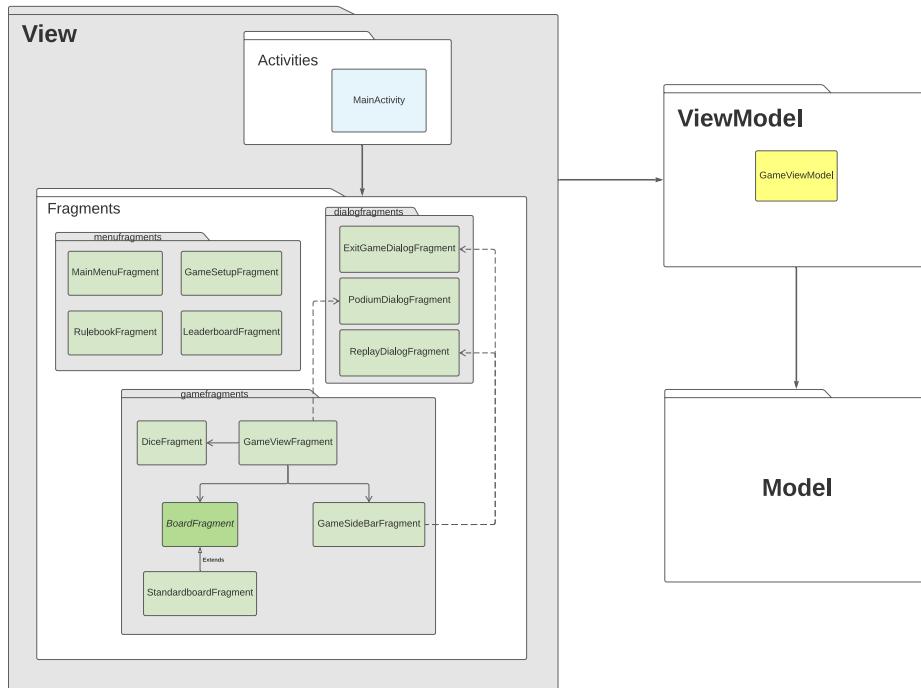
Typiskt sett så fortsätter användaren till GameSetupFragment där ett nytt spel kan startas. När inställningarna för spelet är klart så skapas och initialiseras modellen med den givna informationen och därefter navigerar programmet till GameViewFragment. I sin tur initialiseras den de nödvändiga fragmenten 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.



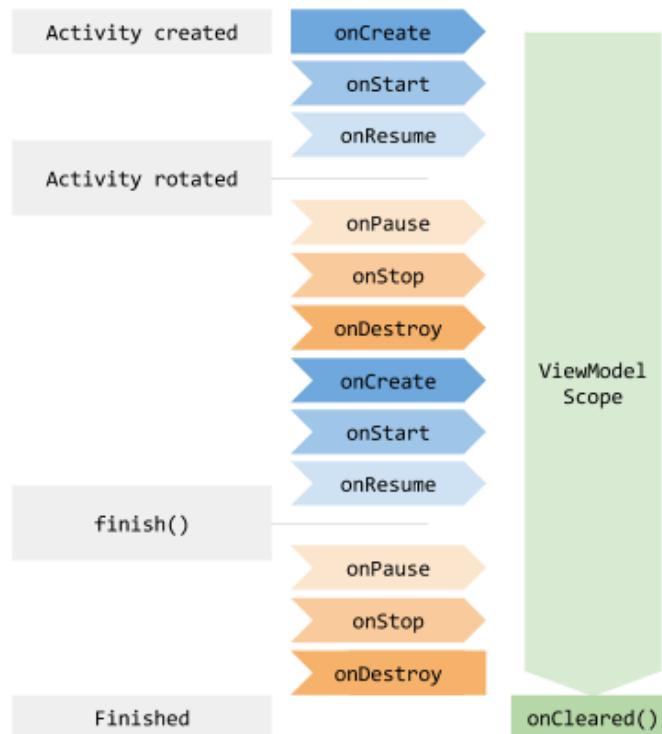
Figur 1: Designdiagram

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 livscykelmedvetet 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 jämfört 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 vy-modell. Datan för dessa vyer sparas temporärt som attribut i deras respektive vy-element. Detta sker automatiskt genom Android ramverket. Den enskilda vy-modellen kommunicerar direkt med modellen endast genom Game-klassen där Game fungerar både som en logisk enhet och en fasad. Detta betyder att vymodel-len 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 båda 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:

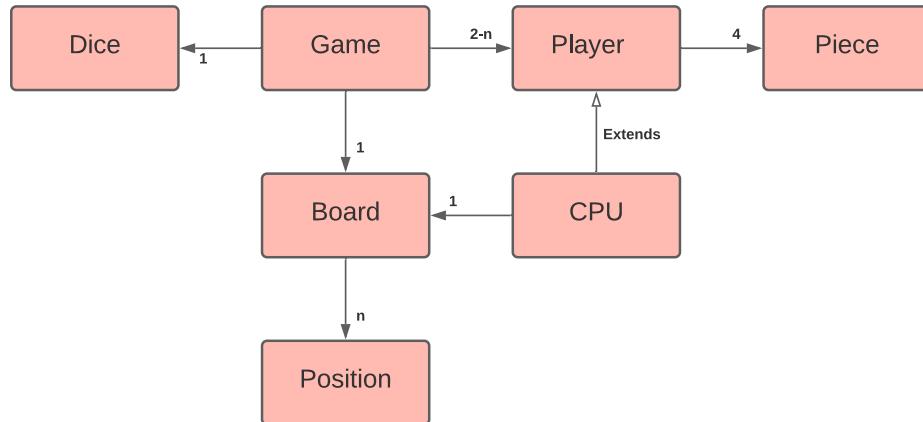
- NotImplementedException - När klienten försöker använda en funktion som ska finnas men som inte ännu är implementerad så kastas detta undantag.
- NotFoundException - Kastas när ett värde eller objekt sökes men inte kunde hittas.

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

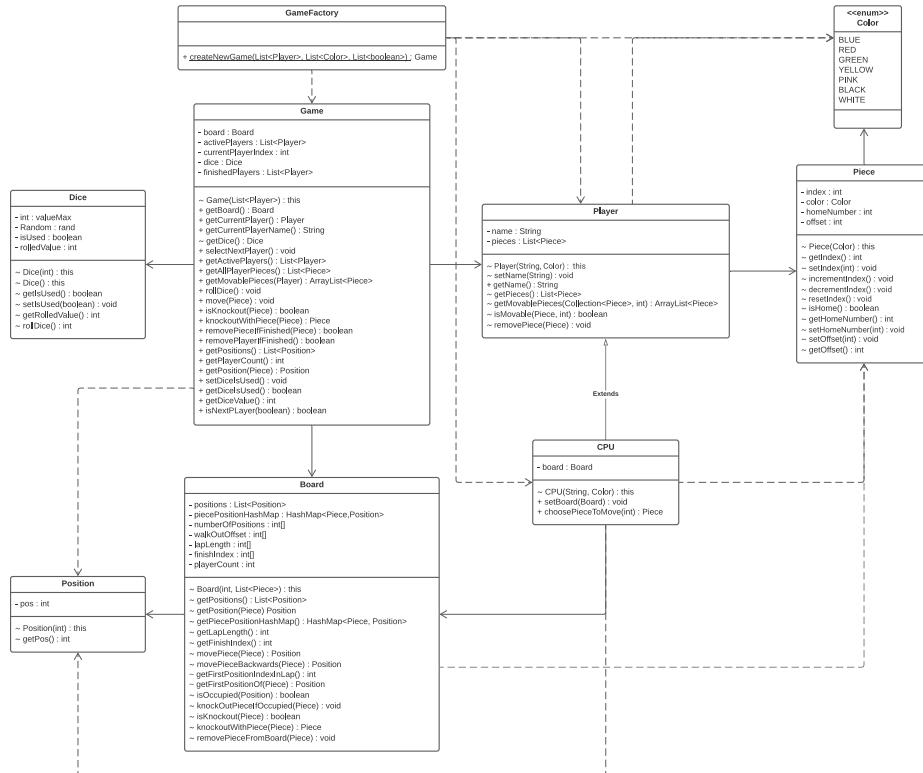
För varje klass i domänmodellen finns en motsvarande klass i designmodellen, samt några ytterligare hjälpklasser. På så vis är designmodellen starkt relaterad till domänmodellen. Designmodellen har dock starkare kopplingar mellan vissa klasser än vad domänmodellen förmedlar, i synnerhet relationen mellan 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

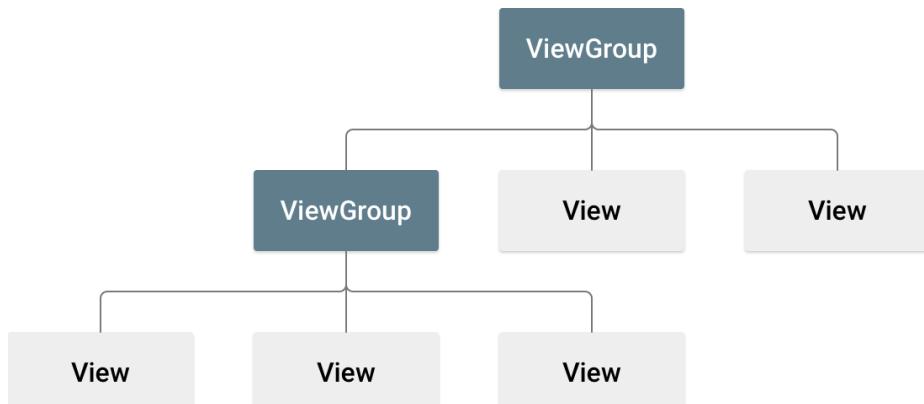
3.3 Vyn

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 kallas 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 innehåller 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

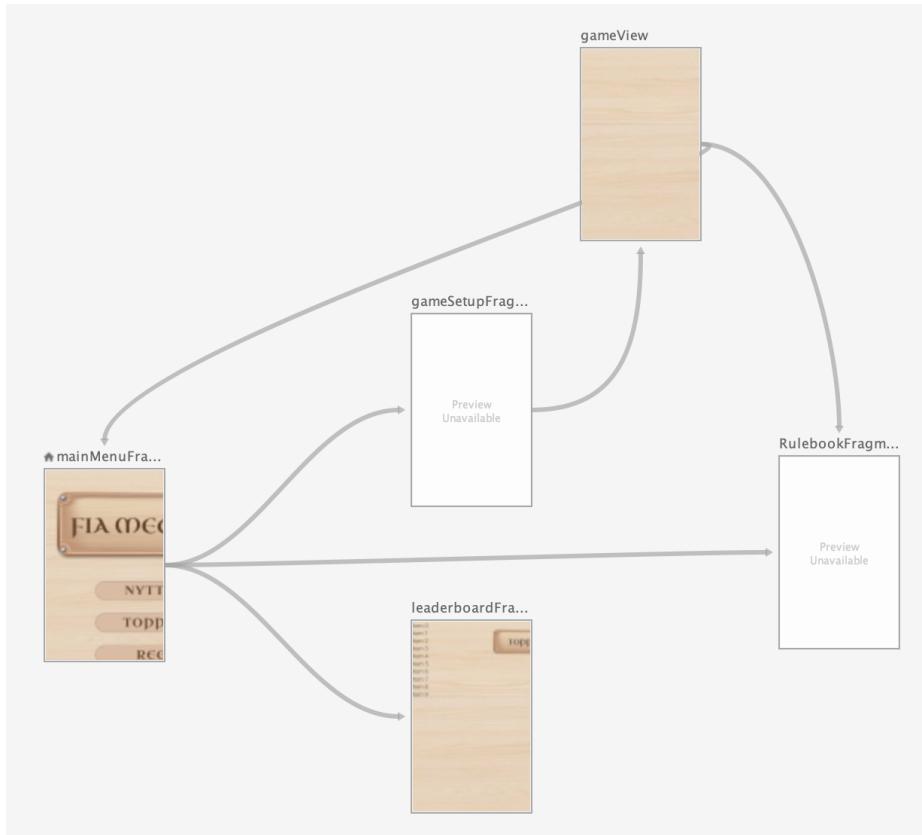
En aktivitet i Android representerar en skärm i användargränssnittet medan ett fragment även kan representera en modulär del av skärmen som kan bytas ut och ersättas med andra fragment. Flexibiliteten med fragment gör att användandet av dessa åstadkommer ett mer modulärt och återanvändbart användargränssnitt. 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 kollar vyn på en lämplig metod hos vy-modellen som i sin tur förmedlar förfrågan till modellen. Om en händelse inträffat som påverkar gränsnittet, 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 observeras 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 livscykler. 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 försätts 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 spelyn stängts ned och sedan skapats om, vilket gjort att datan förstörts. Som nämnts i avsnitt 3.1.1 är detta något som lösas 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 används.

3.5.1 MVVM

Hela applikationen bygger på designmönstret MVVM, vilket har beskrivits uttörligt 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. Overallt 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.

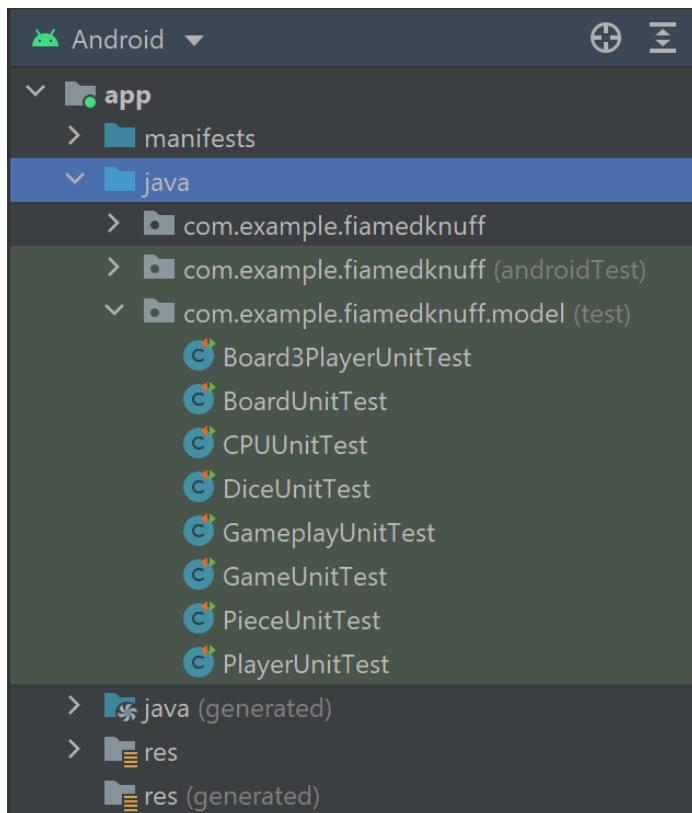
För utökningen att kunna stänga ner ett pågående spel och ta upp det senare behöver det aktuella tillståndet av spelet att sparas, så att spelet vid återstart fortsätter från samma läge som när det pausades eller stängdes ner.

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 allt 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 födröjning vilket gör att det går snabbare än GUI:t kan uppdatera.
- CPU:ns tärningsslag är opålitigt och kan påverka efterkommande spelares drag.
- Om en spelare går ut med en pjäs och sedan går in och ut ur regelboken så kraschar applikationen.
- 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).

Peer-Review av Handymen, Alchemy Defense

Generellt sett är det en väldigt bra kodstruktur och väldokumenterat. MVC följs mycket väl och koden är lättförståelig. SOLID-principerna efterföljs bra i stort med undantag för "Open-Closed Principle". Se kommentarer nedan för förslag på förbättringar.

- Följer design och implementation designprinciper?
 - Håller projektet en konsekvent kodstruktur? Ja.
 - Är koden återanvändbar? **Ja till stor del. Problem: se punkt 1, 2, 6, 8, 10 & 13.**
 - Är den lätt att underhålla? **Ja till stor del. Problem: se punkt 1, 2, 5, 6, 7, 10 & 13.**
 - Kan man enkelt lägga till och ta bort funktionalitet? **Ja till stor del. Problem: se punkt 1, 2, 5, 6, 7, 9, 10 & 13.**
 - Används designmönster? Ja. Singleton, Module, Observer, MVC, (Marker Interface Pattern, används inkorrekt, se punkt 12).
- Är koden dokumenterad? **Mestadels, saknas javadoc i t.ex. ConcreteBoard.java.**
- Används bra och tydliga namn? Ja.
- Är designen modulär? Finns det några onödiga beroenden? **Ja, designen är modulär men det finns onödiga beroenden. Problem: se punkt 1, 6, 7, 8 & 13.**
- Använder koden korrekta abstraktioner? Ja, i nästan samtliga fall. **Problem: se punkt 1 & 9.**
- Är koden bra testad? **Nej, täckningsgraden av testerna kan förbättras, se punkt 4.**
- Bidrar koden till en säker användning gällande information? Finns problem gällande prestanda? **Inget som är relevant.**
- Är koden enkel att förstå? Har den en MVC-struktur och är modellen isolerad från de andra delarna? **Koden är lättförståelig och följer MVC-strukturen nästan till fullo, men problem finns, se t.ex. punkt 8.**
- Kan designen eller koden bli förbättrad? Finns det bättre lösningar? **I stort sätt är programmet välstrukturerat och genomtänkt, men vissa saker kan tänkas över och förbättras. Se nedanstående punktlista.**

-
1. Klassen Tower ger möjlighet för bra kod-återanvändning men ingen av dess subklasser implementerar eget beteende vilket innebär att de är onödiga. I nuläget kan alla subklasser skapas som varianter av superklassen Tower (förutsatt att den inte vore en abstrakt klass). Det kanske är tanken att det ska implementeras senare? Däremot kan kanske ett Tower bättre konstrueras med hjälp av komposition och ge den egenskaper via "dependency injection". Underlättar vid upgradering, ändring av attackmönster, återanvändning. Undvikar klassexpllosioner.
 2. TowerPrices specificerar köp- och säljpriser för alla tornen. Gör dock så att ett pris för ett torn är frånkopplat från själva tornet. Finns för och nackdelar men om man skulle vilja uppgradera ett torn (finns inte nu verkar det som) så är prisen redan satta och det blir knepigt att hålla koll på vilket pris varje torn ska ha.

3. *TowerPrices* kan möjligtvis bara hålla sina “private static final ints” som publika istället och låta komponenter använda dem genom klassen direkt. Ingen metod för det behövs utan *TowerPrices* agerar endast som en dataklass. Ett pris skulle då nås genom att skriva `TowerPrices.RED_TOWER_BUY_PRICE` för ett rött torn.
4. Inte tillräcklig täckningsgrad på tester. Programmet uppnår endast 56% täckningsgrad i testerna av modellen.
5. Designmönstret Singleton används i klassen *Player*. En spelare kommer alltid skapas med startvärdena 100 guld och 100 hp och det blir krångligt om man skulle vilja lägga till någon slags förmån eller köra i ett speciellt läge där spelaren exempelvis vill skapas med 10000 hp och 1 miljon guld. Går visserligen att använda metoder för att öka guld och hp efter skapelse men blir knepigt vid flerspelarläge. Känt för singleton är att JUnit-tester kan bli svåra att utföra då mönstret introducerar starka couplings och oönskade sidoeffekter. Mer om detta går att läsa om [här](#).
6. Klassen *Tile* har två instanser *Tower* och *Foe*, men en tile kan endast ockuperas av endera, alltså har den en instans som klassen inte “behöver” vid ett givet tillfälle. Både *Tower* och *Foe* implementerar *BoardObject* och det borde vara möjligt att låta *Tile* ha ett *BoardObject* istället. Underlättar även vid implementation av fler typer, säg någon slags blockad.
7. *Tile* använder “reaching through” men borde kanske snarare ha en metod “`getTowerDamage`” istället för att metoden `damageFoe()` kallar direkt på metoden vilket skapar ett beroende direkt från *ConcreteBoard* (klassen `damageFoe()` finns i) till *Tower*.

```
private void damageFoe(Tile cellTower, Tile cellFoe) {
    int damage = cellTower.getTower().getDamage();
    cellFoe.getFoe().takeDamage(damage);
}
```

8. “Interface” *BoardObject* som många klasser implementerar har krav på en metod som hämtar klassens sökväg till dess bild vilket bryter mot MVC mönstret. Modellen får en direkt koppling till vyn.

```
public interface BoardObject {
    String getImageFilePath();

    BoardObjectType getBoardObjectType();
}
```

Dessutom ska modellen vara överförbar till vilket GUI-ramverk som helst och där behövs eventuellt ingen sökväg till den klassens bild. Dessutom är dessa sökvägar “private final Strings” vilket innebär att om man skulle vilja ändra hur något ser ut när programmet körs så blir det väldiga problem. Exempelvis om man skulle vilja flasha rött när en skurk blir skadad eller om ett torn uppgraderas.

9. *Pathfinder* skulle kunna abstraheras. *ConcreteBoard* har just nu en *Pathfinder* som kan utföra “breadth-first search” (BFS) som endast har två publika metoder: `generateNewPath` och `blocksPath`. Låt istället *Pathfinder* vara en “interface” som definierar dessa två metoder och låt *ConcreteBoard* ha en attribut av den. Då kan

konkreta klasser av typen Pathfinder konstrueras som exempelvis BFSPathfinder, DFSPathfinder (depth-first), AstarPathfinder (A^*) osv. Kan ses som överflödigt men skulle kanske bli nödvändigt om man skapar ett större tilegrid / mindre rutor. Liknande funktionalitet bör kunna uppnås med Strategy Pattern.

10. I klasserna *TowerFactory* och *TowerPrices* används många "switch"-satser vilket inte är optimalt då det bryter mot designmönstret "Open Closed Principle". Ska det exempelvis utökas med fler torn blir det svårt eftersom de nya tornen måste läggas in på alla ställen. Glöms ett ställe märks inte detta förrän programmet redan körs och det blir ett exekveringsfel.
11. Inte särskilt användarvänligt t.ex. när det kommer till att högerklicka för att ta bort objekt. Man skulle isäfall vilja få den infon på något sätt. Svårt att hänga med i spelet p.g.a. långsam uppdateringsfrekvens.
12. "Interface" *Foe* beskrivs använda sig av designmönstret "Marker Interface Pattern" men implementerar inte detta korrekt eftersom *Foe* definierar metoder för de som implementerar interfacet. För att få önskat beteende som "Marker Interface Pattern" ger kan istället "annotations" användas i Java men så som *Foe* är i nuläget behöver den ingetdera utan kan helt enkelt vara ett "vanligt" "interface".
13. Delar av programmet skulle nyttjas av "Dependency Injection"-designmönstret (DI). Exempelvis i klassen *ConcreteBoard* som just nu har hårdkodade värden för dess *TileGird*-instans och skapar objekt av andra klasser i sin konstruktör. Vill man i nuläget ändra storleken på brädet så måste man ändra på klassen i sig. Genom DI eller flera konstruktörer så görs kopplingen mellan *ConcreteBoard* och de andra klasserna svagare samt det får en mer flexibel och återanvändbar klass.