# 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.

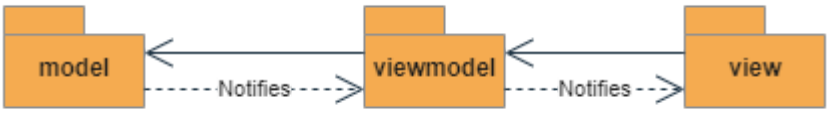
### 1.1 Definitioner

- **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.

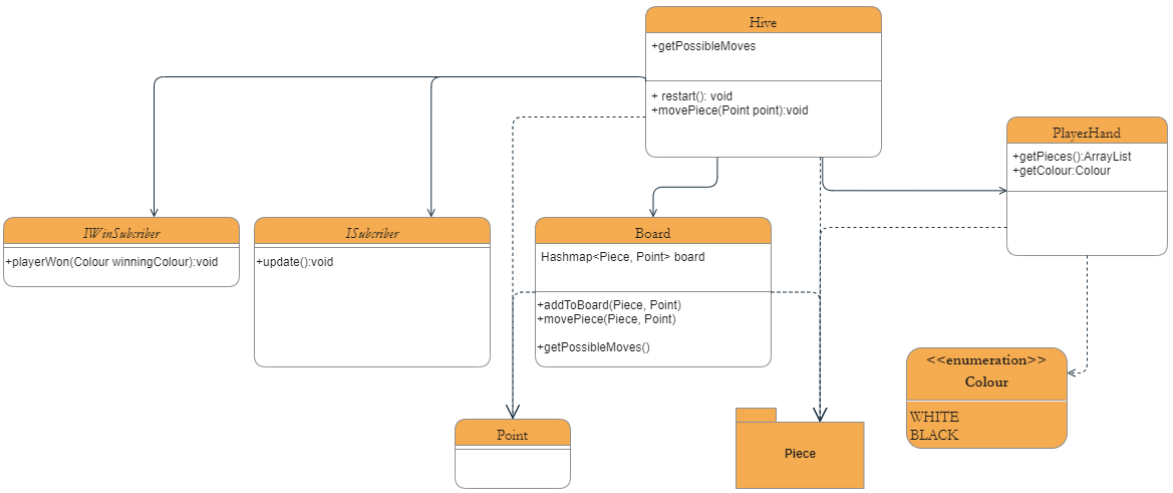
I MVVM- mönstret så fungerar vy-modellen som en mellanhand mellan vyn och modellen och det är vy-modellens jobb att hantera de klickningar som vyn registrerar, och att presentera datan som vyn sen ska visa. För att vy-modellen ska vara livscykeloberoende och kunna kopplas till olika vyer så får vy-modellen 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.

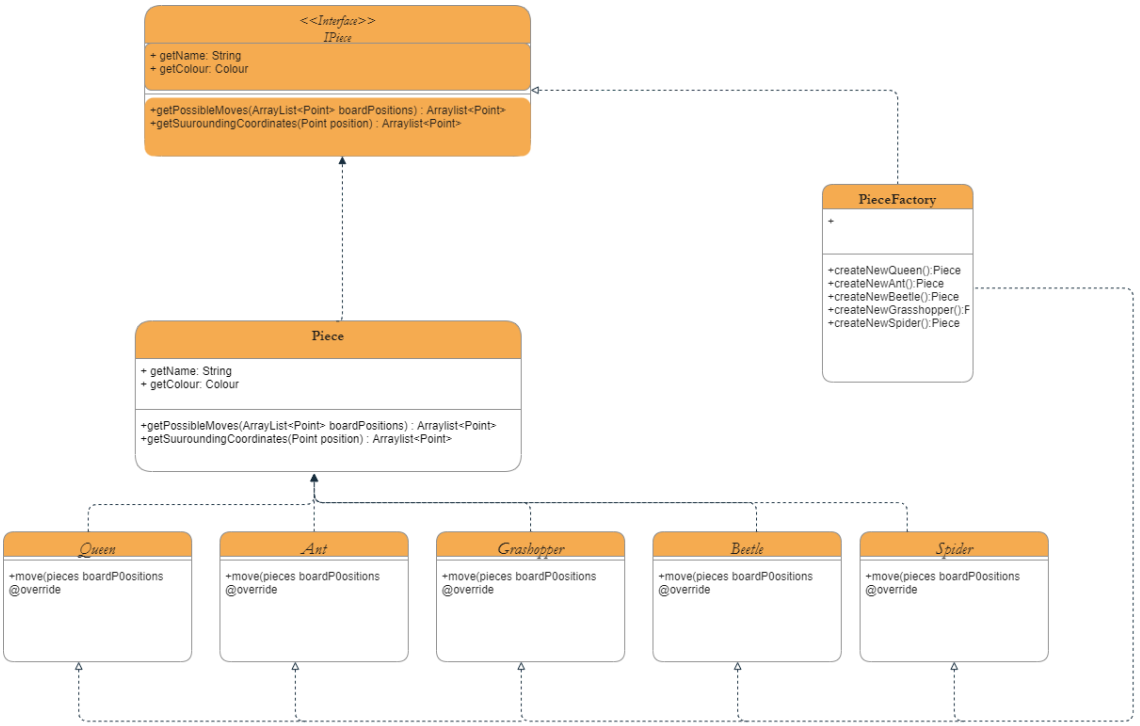
Modellen har inte heller några beroenden på vyn eller vy-modellen. Men för att vy-modellen ska få reda på när något i modellen har uppdaterats har designmönstret observer använts. Detta innebär att modellen innehåller en lista med subscribers som implementerar interfacet Isubscriber och när den uppdateras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

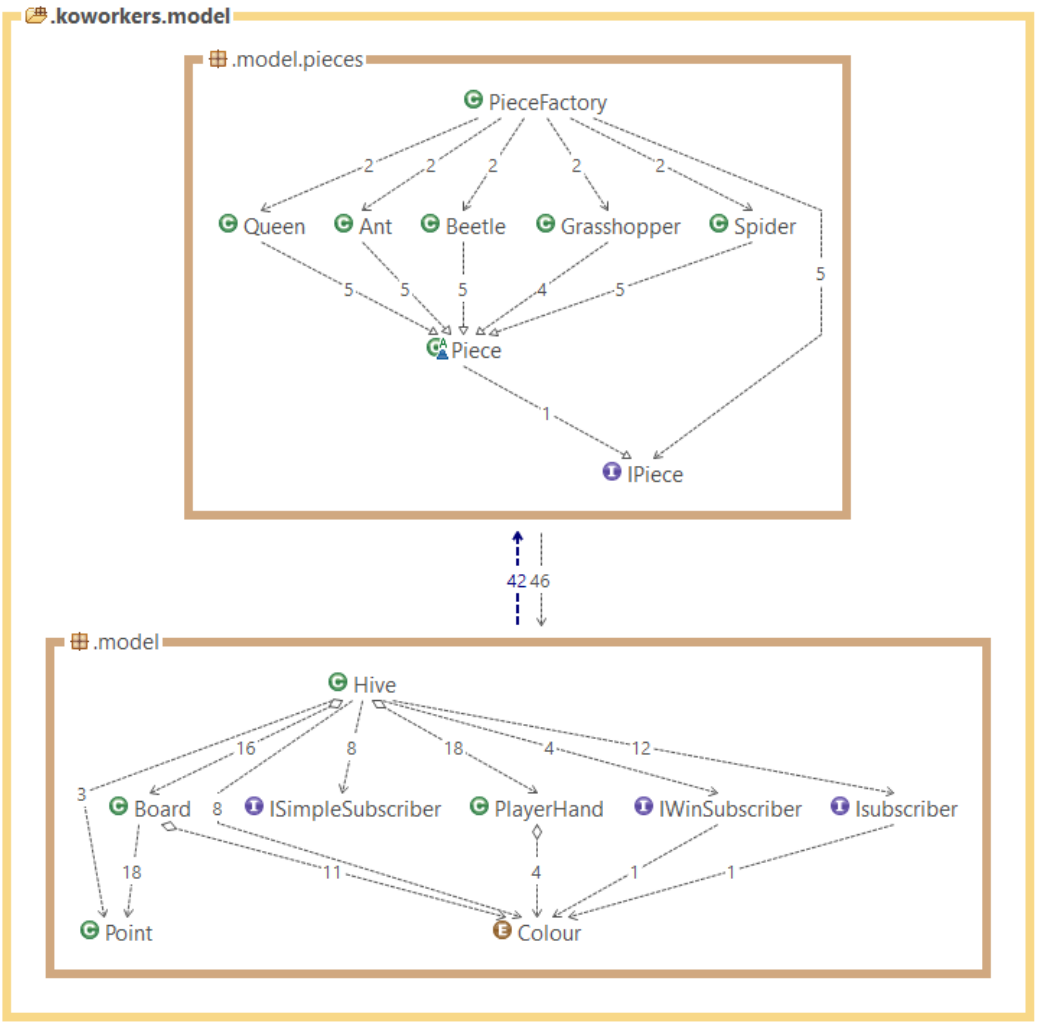
## 5.3 Analytiska verktyg

### 5.3.1 PMD

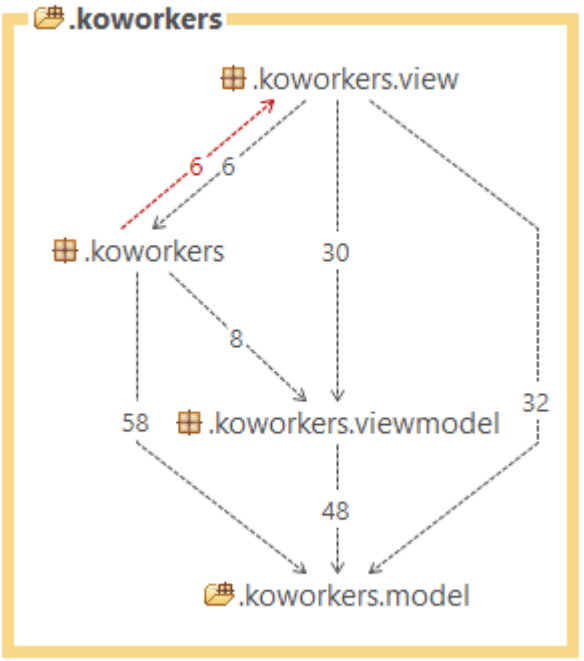
PMD är program som analyserar kod och hittar vanliga fel. Här används en plugin i Android Studio, PMDPlugin, för att analysera koden. När PMD med standardinställningar körs på projektet fås 633 problem, men majoriteten av dessa är antingen inte relevanta eller obetydliga problem. Exempel på problem som 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.