Snake App – Rapport

DAT255 - Software engineering project

Jesper Sjövall
Martin Sonesson
Alesandro Sanchez
Rickard Persson

Index

Index

Licens

1 Introduktion

Det här är den slutgiltiga rapporten av vårat Snake App projekt.
Projektets mål var att ta fram en korrekt fungerande prototyp av en Android app.

Vi vale att göra vårat egen version av det klassiska spelet 'Snake'. För att vårat spel skulle urskilja sig från övriga Snake spel så beslöt vi oss för att utnyttja de funktioner som G-sensorn erbjuder. G-sensorn är en sorts accelerometer som finns i de flesta Android smartphones (om inte alla) som kan mäta rörelser och känna av hur man håller telefonen.
Vi ville ha ett snake Spel där man styrde ormen helt och hållet genom att att luta och vrida på telefonen. Ingen d-pad eller touchknappar används under spelets gång.

1.1 Mål med designen

Våra huvudsakliga mål var att ta fram ett fungerande Snake spel åt Android telefoner som styrs med G-sensorn, såsom förklarat i föregående stycke.
Vi ville också ge användaren möjlighet att spela på olika nivåer där varje mål kan ha en egen uppbyggnad och värden som justerar svårigheten.
För att användare ska kunna hålla koll på sina framsteg så ville vi även implementera en highscore-funktion, det vill säga att spelet håller koll på de tio bästa resultaten och om man får tillräckligt många poäng efter en spelomgång får man skriva in sitt namn och spara denna information i en highscore-databas.

1.2 Definitioner, akronym och förkortningar

Vi använder bl.a begreppet ***databas*** när vi syftar på ett lagringsmedel som kan spara data på ett förbestämt sätt, där enklaste exemplet är en fil på systemet.
Vi använder även begreppet ***IC*** när vi syftar på en Java-interface klass för att kunna särskilja mellan interface och faktiskt klass. Tex så finns det *GameEngineIC* som är interfacet för Spelmotorn.

1.3 Referenser

Hela projektet återfinns på följande adress:

```
1 http://snake-app-project.googlecode.com/
```

2 Koduppbyggnad

Vi har redan från början i detta projektet styrt programflödet till att blir ett modulbaserad programflöde vilket både har sina för och nackdelar på många punkter.

Några av de fördelarna som vi ser med ett och använda en modulbaserad programflöde är att om man har gjort ett gott förarbete med tydliga *API* (interface) så är det i möjligt för flera personer samtidigt jobba med olika moduler utan och skapa onödigt dubbelarbete, konflikter mellan olika lösningar och moduler.
Nackdelen med och använda ett modulbaserad programflöde är att det är svårt i förväg och veta hur stora varje modul kommer och blir och ett tydligt exempel på detta kan vara den modulen som vi från början skulle samla in system event försvann helt då den blev den del av de grafiska modulerna i programmet.

Men den verkligt stora fördelen med och använda ett modulbaserad programflöde är hur enkelt det är och byta ut en särskild modul då den har en väldefinierad och relativt litet ansvarsområde och väldefinierad sätt och kommunicera med andra moduler.

Ta tex grafiken, ifall vi vill göra om vårt program till och fungera på en vanlig dator i stället för Android så skulle det har räckt och skrivit om GUI och hur man styr spelet då dessa två delarna i teorin är det enda som är strikt kopplad till Android plattformen.
Faktum är att vi från början utvecklare spelmotorn mot en vanlig *JAVA SE* plattform, och det enda som behövdes göra för och anpassa den till Android plattformen var och ändra 3 rader kod då det var ett par metoder som inte fanns i Android som vi hade använt oss av.

2.1 Allmänt

Vi kommer här i korthet försöka ger en överblick över de olika modulerna i programmet.

- Interface
  Detta är egentligen ingen riktig modul utan snarare en samling av regelverk hur olika moduler får och skall prata med varandra.

- GameEngine
  Spelmotorn ansvara för den matematiska modellen som användas för och beräkna spelets gång under ett pågående spel och ligger till grund för det som skall visas för slutanvändaren i GameUI

- GameUI
  Är den grafiska avbildningen av det som GameEngine vet som en matematisk modell.

- Highscore Databasen
  Är den modulen som tillhandahåller tjänster för att lagra och sortera de bästa resultaten som en användare har fått på enheten.

- Level Databasen
  Tillhandahåller tjänster för och lista alla nivåer som finns inlagda i programvaran, samt läsa in vald nivå till ett standardiserad format som både GameEngine och GameUI förstår.

- MotionDetector
  Tillhandahåller IO tjänst för att erhålla information om hur användaren vill integrera under ett pågående spelomgång, så som tex ifall använda önska svänga höger så upptäcker MotionDetector det och förmedlar detta vidare till de tjänster och moduler som är intresserad av det.

- Master Controller
  Spindel i nätet, som kan lista alla de olika tjänsterna som finns i de olika modulerna som är kopplade till programvaran.

Sedan så tillkommer det några gemensamma verktygsmoduler för att underlätta kommunicera mellan en eller flera moduler.

2.2 Lager

Vi har valt för enkelhetens skull och lägga nästan alla modulerna i samma nivå, då vi inte har några moduler som som är tillräckligt stora för det skall vara någon meningen och bryta ner dem till nya undermoduler, utan istället blir klasser.
Som faktisk kan se som små självständiga moduller utan större vetskap om som omvärld. Men här har vi valt att varje programmerare kan och skall utforma varje klass modul efter egen ide och tanke så länge de följer de gemensamma API och interface där det efterfrågas.

2.3 Uppdelning till subsystem

Programmet är uppdelat enligt model-view-controller (MVC), vilket lämpade sig för detta projekt då de olika programdelarna enkelt kunna uppdelas enligt denna struktur. Själva spel modellen utgjordes av paketet *gameengine* där klassen *GameEngine* utgör huvudklassen för modellen medan klasserna *LevelEngine* och *PlayerBody* representera modellen för enskilda bannor samt spelarkroppen.

Modelen använder sig även av paketet *leveldatabase* för att få tillgång till de färdiga gjorda level filerna, när en bana läses kommer denna att representeras av klassen *XMLLevel* som implementerar *LevelIC*.

För att delen av programmet som utgör *viewer* ska kunna få tillgång till modellen har *GameEngine* implementerat interfacet *EnumObservable* och genom detta skickar modellen regelbundet uppdateringar till de klasser som registrerat sig för dessa.

Själva kontroll delen av MVC utgör i huvudsak av två paket nämligen *mastercontroller* vilket innehåller klassen *ControlResources* vilken utgör kärnan i programmet genom att initiera olika delar av programmet samt gör dessa tillgängliga för andra delar av programmet vilket möjliggör kommunikation mellan dessa delar.

Den andra delen av kontrollenheten återfinns i paketet *motiondetector* och klassen med samma namn i detta paket. Genom dessa klassen hantera händelser från telefonens rörelsesensorer och omforma dessa till information som är kompatibel med programmet, slutligen görs denna information tillgänglig för andra delar av programmet framförallt *gameengine*.

Den slutliga delen av MVC-modelen utgörs av *viewer* delen vilket i detta program utgörs av paketet *gameui* vilken innehåller klassen *GameView* vilken ansvarar för att ge en grafisk representation av modellen, genom att rita ut hinder, äpplen samt ormen.

Till detta finns även ett *highscoreDatabase* vilken utgör ett komplement till MVC-modelen som hantera en lista med de tio bästa resultaten som uppnåtts i den specifika applikationen. Utöver detta finns klassen *Storage* vilken används för att spara och hämta data till och från telefonens minne.

2.4 Dependency analys

Vår applikation använder sig av interface vilket medför att det upplevs enligt bifogad bild 1 att många beroende finns till paketet *interfaces* och de interface som detta paket innehållet. Genom detta val av strukturering minskas den direkt kopplingen mellan de olika klasserna. I denna genomgång av beroende mellan olika klasser kommer de faktiska klasserna omnämnas istället för interface som dessa klasser implementera.

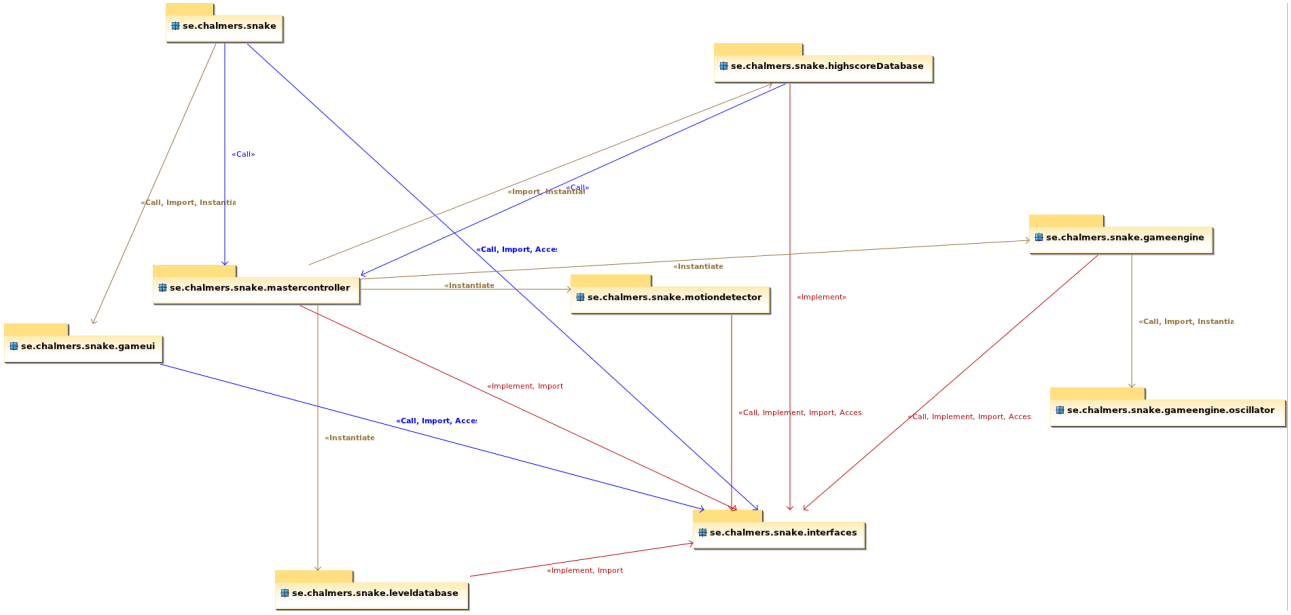

Illustration 1: Se bilaga nr 1 för större bild

Det finns två paket som oberoende av andra delar av programmet nämligen util, interface samt *motiondetector*, därefter kommer *leveldatabase* där klassen *XMLLevel* beroende av *REPoint* och *XYPoint* vilka används i konstruktionen av banorna.

Klassen *GameEngine* har förutom de interna klasserna i paketet *gameengine* som beroende av klassen *XMLLevel* då spelmotorn av naturliga själ behöver banor, för att få tillgång till banorna finns beroende till klassen *LevelDatabase*. Det finns också beroende av klassen *EnumObservable* för att spelmotorn ska kunna skicka information till observers. Till detta finns ett minimalt beroende av klassen *ControlResources* i *mastercontrol* för att spelmotorn ska veta storleken på bildskärmen.

Paketet *gameui* och därmed klassen *GameView* har beroende till *GameEngine* för att få uppdateringar hur spelet fortgår, det tillkommer *REPoint* för att indikera positionen för de olika elementen i spelet.

Den enskilda klass som är mest beroende av andra klasser är *ControlResources* i *mastercontrol* vilket beror på att denna klass fungera som navet i vår applikationer och möjliggör att de olika komponenterna kan sammankopplas. Klassen är beroende av *GameEngine*, *HighscoreDatabase*, *LevelDatabase*, *XYPoint*, *MotionDetector* och *Storage*.

Klasserna i paktet *highscoreDatabasen* som hantera de tio bästa resultaten, de två klasserna som ingår här är *Highscore* som representera ett resultat och saknar beroende till andra klasser i applikationen. Den andra klassen i paketet är *HighscoreDatabase* som hantera lagringen och läsning av alla highscore, genom detta finns beroende till *Storage*, *ControlResources* och *Highscore*.

Sedan finns det ett antal klasser som extendar *Activity* nämligen de fyra klasserna i paketet Snake, *GameActivity* som initiera själva spelet vilket för att denna klass har beroende till de komponenter som är involverade i detta, alltså *GameEngine*, *XMLLevel*, *ControlResources*, *EnumObservable* och *EnumObserver.*

StartActivity hantera själva huvudmenyn och har därigenom endast ett fåtal beroende på mycket hanteras av *ControlResources* vilket *StartActivty* har beroende till, det finns även beroende till *HighscroreDatabase* för att användaren ska kunna få tillgång till denna lista av de bästa resultaten.

HighscoreActivity hantera databasen med samma namn och har därför beroende till klassen *Hightscore* database samt kontrollenheten för applikationen *ControlResources*. Slutligen finns *SelectLevelActivity* vilken hantera historiken om vilka banor som användaren har kört, genom detta finns beroende till *LevelHistory* och *ControlResources*.

2.5 Problem vid samlöpning (parallell programmering)

Vi förlitar oss på Androids metoder att dela upp processtid mellan de olika processerna som körs på sådant sätt att inga delar av programmet lider av svält.
Vi finner inget behov av att aktivt påverka hur processortiden fördelas då programmet är top-programvara som inte behöver ta hänsyn till underliggande programs resursbehov.

2.6 Statisk datalagring samt åtkomstbehörighet

Vi har använt oss av Android plattformens inbyggda system för statisk datalagring där Android plattformen hanterar åtkomstbehörighet av den sparade data. Vi har valt att spara data så enbart så enbart vårt program kan komma åt den data och inställningar som vi har gjort genom att sätta följande inställningar i Android när vi sparar data:

```
1 android.content.Context.openFileOutput(
2     fileName,
3     android.content.Context.MODE_PRIVATE
```

```
1 );
```

Själva data sparar vi som Java serializing format.

2.7 Boundary conditions

Spelet startas genom att klicka på dess genväg, detta är standard i Android. När applikationen startas så körs vår första *StartActivity*. Denna initierar utseendet samt skapar vår *ControlResources* som tillhandahåller våra resurser, bland annat spelmotorn, *HighscoreDatabasen*, inlästa nivåer samt rörelsesensor. Om spelaren väljer att avsluta applikationen i denna activity så avslutas den genom att anropa classens ”finish” metod som är det rekommenderade sättet att avsluta activities på.

Det bästa sättet att hantera vad applikationer gör i Android är genom att dela upp funktioner i activities. När appen startar så anropas dess main activity, i vårt fall *StartActivity*. Denna kan sedan starta andra activities genom att skapa ”intents” som skapar den nya activity och skickar en dit, när denna avslutas så återkommer man till den activity som skapade den.

Själva spelet körs genom att starta vår *GameActivity* (när spelaren väljer 'New game'). Denna hämtar efterfrågade resurser från *ControlResources* och startar spelet. När spelet är över och man har tillräckligt med poäng kommer man till våran *HighscoreActivity*, annars får man möjligheten att börja om eller att sluta. Det senare leder till att *GameActivity* avslutas När den väl är avslutad kommer man tillbaka till *StartActivity*.

Man kommer till *HighscoreActivity* när spelaren fått tillräckligt med poäng och valt att spara sina poäng. Man skickas hit från *GameActivity*. Denna tar *HighscoreDatabase* från *ControlResources* och sparar spelarens namn och poäng.

En annan activity som kan startas är *SelectLevelActivity*. Den, precis som *GameActivityns*, startas från *StartActivity*. Den tar emot LevelDatabase från *ControlResources* och visar dom för användare så att denne kan välja nivå.

Vi fångar dom flesta fel som kan inträffa (exceptions) och när något inträffar så avslutas activityn och man går upp i hierarkin och återkommer då alltså till föregående activity. Om felet inträffar under *StartActivity* så avslutas appen. Lägg märke till att detta är ytterst ovanligt.

3 Källor

Sida som innehåller ett antal Snake Spel som har varit ide källa.

1
```
http://www.snakegame.net/
```

Sida som beskriver om hur rörelsesensorer fungera.

2
```
http://blog.poweredbytoast.com/getting-tilt-data-on-an-android-phone
```

4 Bilagor

4.1 Dependencies

Ett antal bilder som beskriver kopplingen mellan olika delar av programvaran.

3
```
Bilaga_Dependencies.png, Bilaga_Dependencies.svg
```